SimpleScalar Tutorial

(for tool set release 2.1)

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

Doug Burger
Computer Sciences Department
University of Wisconsin - Madison
dburger@cs.wisc.edu

Tutorial Overview

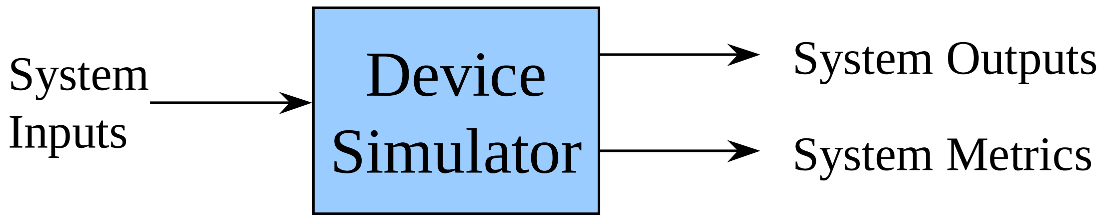
- Overview and Basics
- How to Use the SimpleScalar Architecture and Simulators
- How to Use SIM-OUTORDER
- How to Build Your Own Simulators
- How to Modify the ISA
- How to Use the Memory Hierarchy Extensions
- Limitations of the Tools
- Wrapup

Tutorial Overview

- Overview and Basics
  - ❑ Simulation Primer
  - ❑ SimpleScalar Tool Set Overview
- How to Use the SimpleScalar Architecture and Simulators
- How to Use SIM-OUTORDER
- How to Build Your Own Simulators
- How to Modify the ISA
- How to Use the Memory Hierarchy Extensions
- Limitations of the Tools
- Wrapup

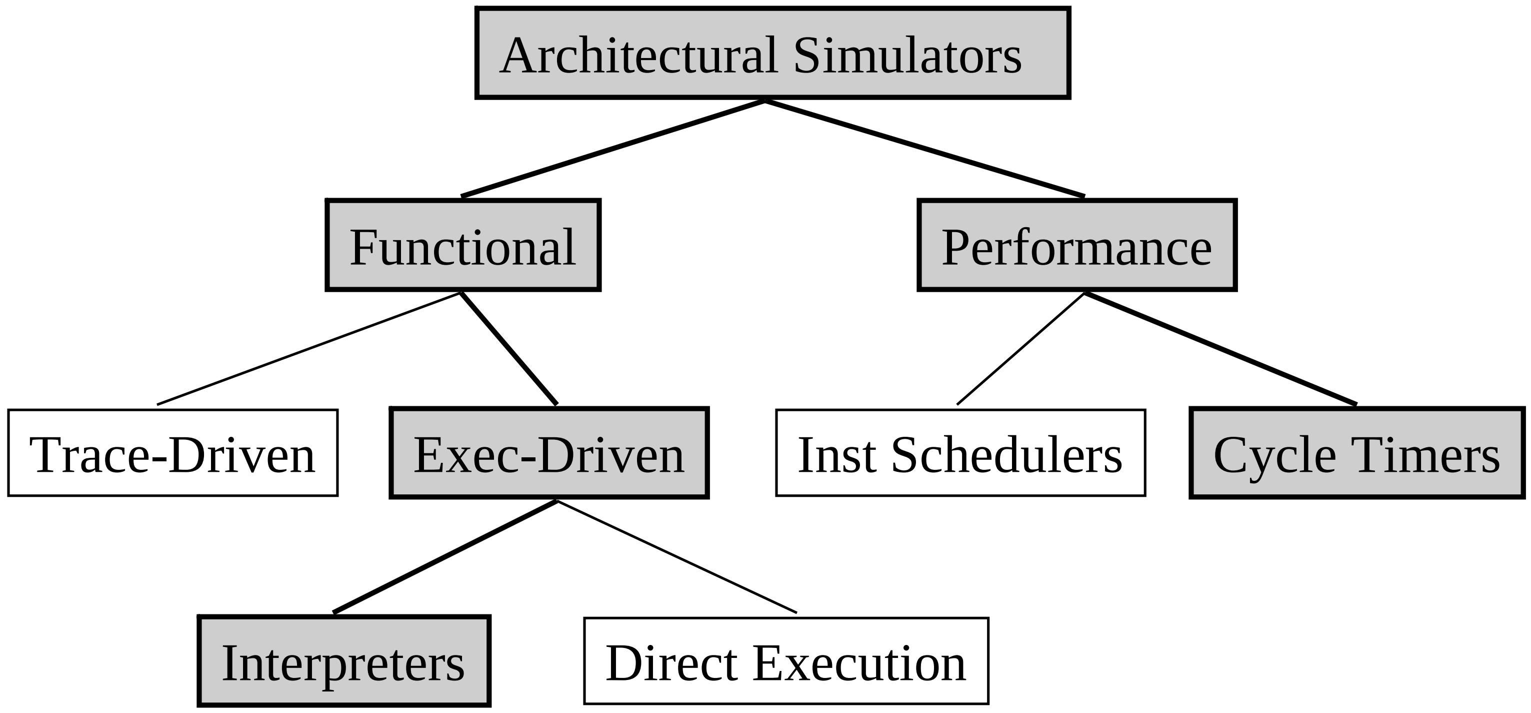
A Computer Architecture Simulator Primer

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



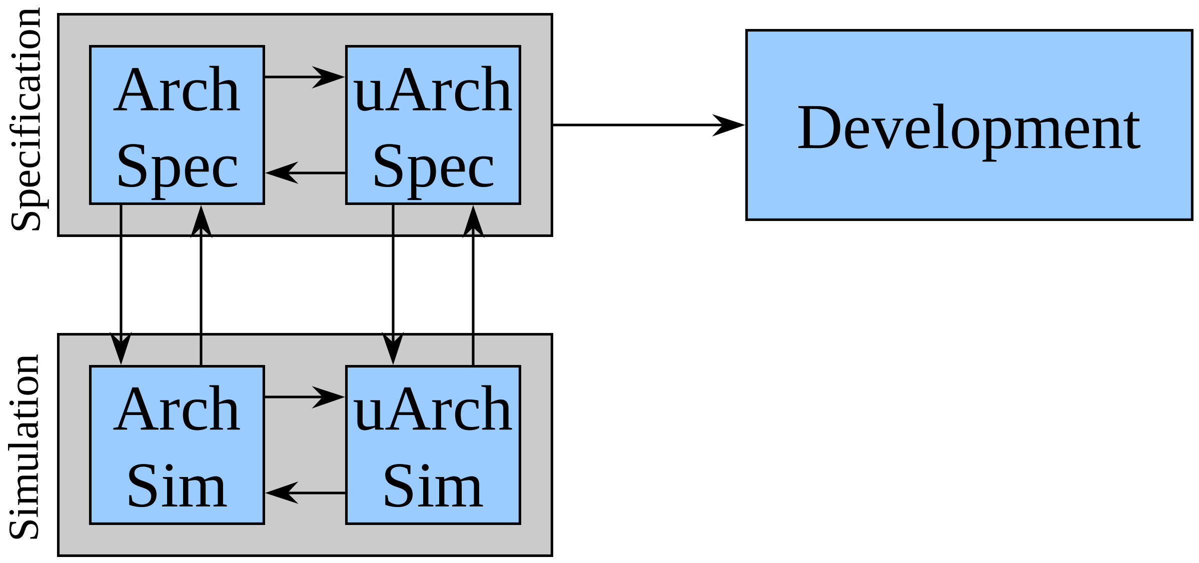
- Why use a simulator?
  - leverage faster, more flexible S/W development cycle
    - permits more design space exploration
    - facilitates validation before H/W becomes available
    - level of abstraction can be throttled to design task
    - possible to increase/improve system instrumentation

A Taxonomy of Simulation Tools



- shaded tools are included in the SimpleScalar tool set

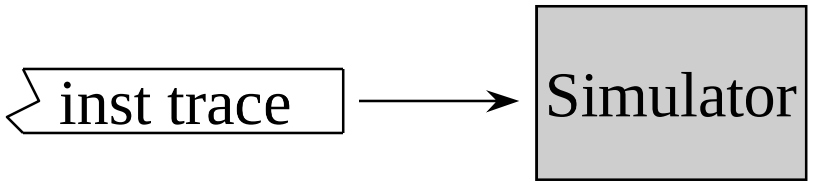
Functional vs. Performance Simulators



- 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

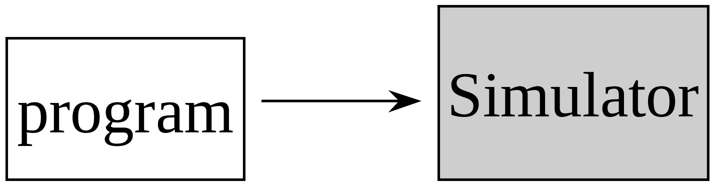
Execution- vs. Trace-Driven Simulation

- trace-based simulation:



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

- execution-driven simulation:

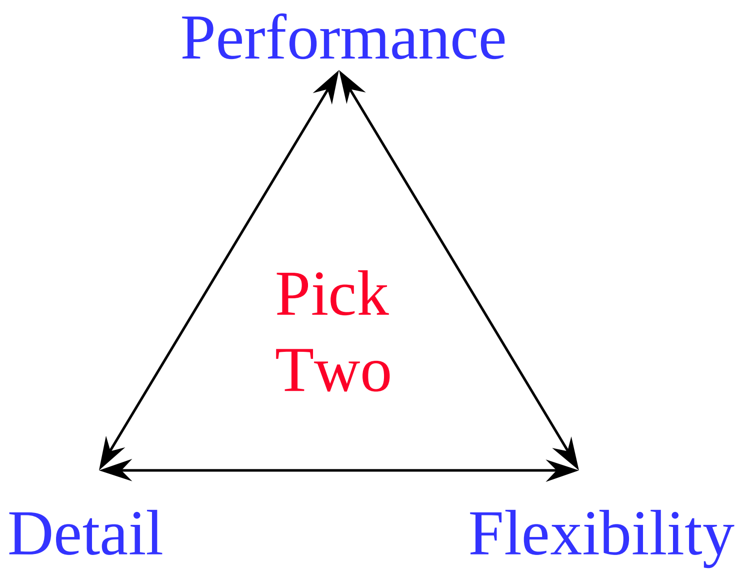


  - simulator "runs" the program, generating a trace on-the-fly
  - more difficult to implement, but has many advantages
  - direct-execution: instrumented program runs on host

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

The SimpleScalar Tool Set

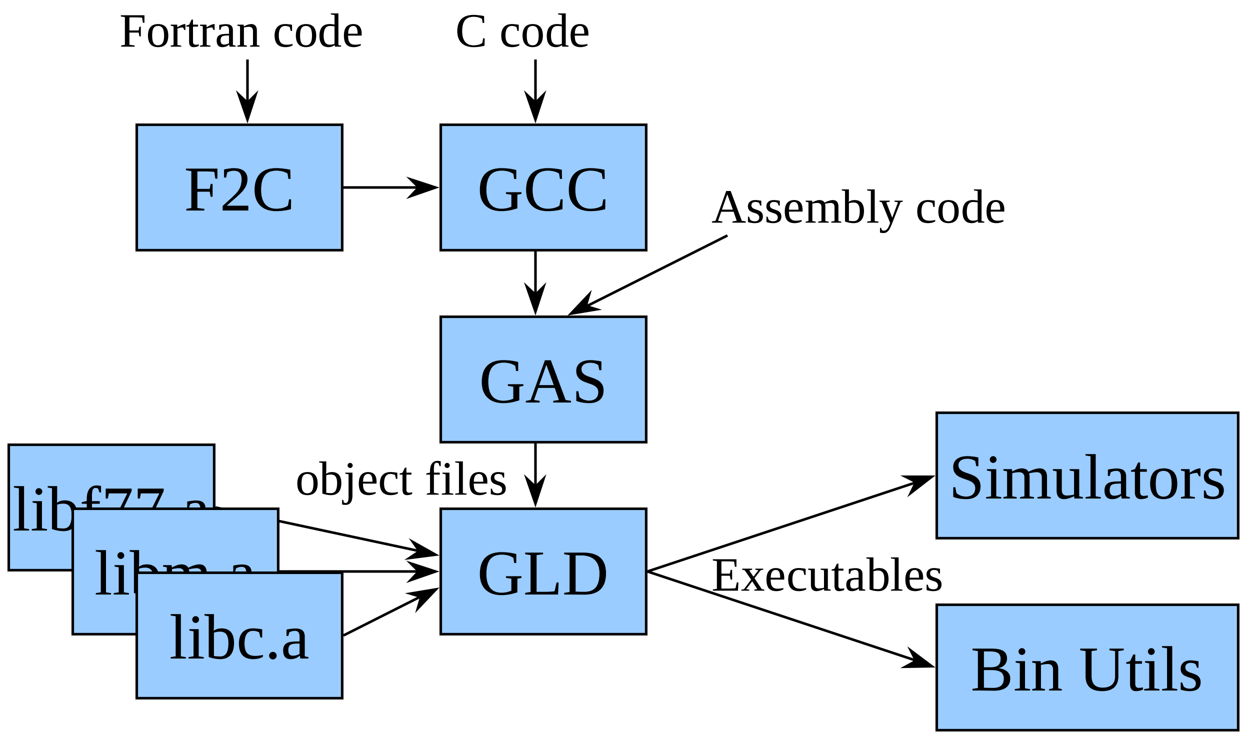
- computer architecture research test bed
  - ❑ compilers, assembler, linker, libraries, and simulators
  - ❑ targeted to the virtual SimpleScalar architecture
  - ❑ hosted on most any Unix-like machine
- developed during my dissertation work at UW-Madison
  - ❑ third generation simulation tool (Sohi → Franklin → SimpleScalar)
  - ❑ in development since ‘94
  - ❑ first public release (1.0) in July ‘96
  - ❑ second public release (2.0) in January ‘97
- freely available w/ source and docs from UW-Madison

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

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

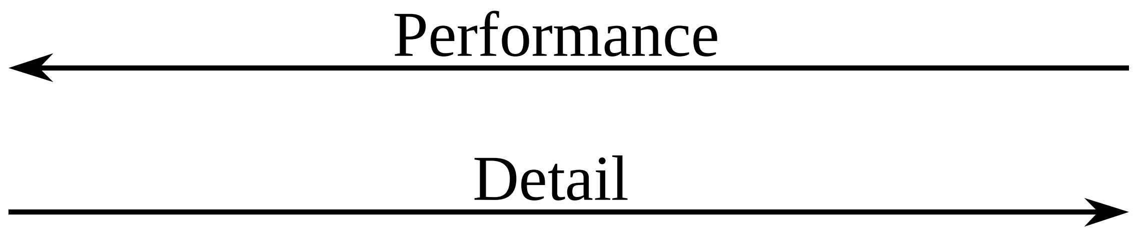
SimpleScalar Tool Set Overview



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

Simulation Suite Overview

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



Tutorial Overview

- Overview and Basics
- How to Use the SimpleScalar Architecture and Simulators
  - ❑ Installation
  - ❑ User's Guide
  - ❑ SimpleScalar Architecture Overview
  - ❑ SimpleScalar Simulator S/W Overview
  - ❑ A Minimal SimpleScalar Simulator (sim-safe.c)
- How to Use SIM-OUTORDER
- How to Build Your Own Simulators
- How to Modify the ISA
- …

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
- can grab SPEC95 binary release in lieu of compilers
- if you have problems, send e-mail to developers or the SimpleScalar mailing list: `simplescalar@cs.wisc.edu`
  - please e-mail install fixes to: `dburger@cs.wisc.edu`
- x86 port has limited functionality, portability
  - currently not supported
  - only works on big-endian SPARCs

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

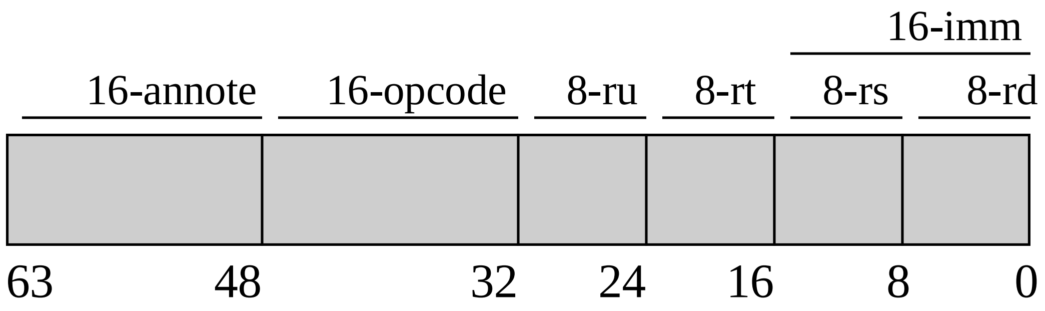
- supported on all simulators:

`-h`	- print simulator help message
`-d`	- enable debug message
`-i`	- start up in DLite! debugger
`-q`	- quit immediately (use w/ `-dumpconfig`)
`-config <file>`	- read config parameters from `<file>`
`-dumpconfig <file>`	- save config 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, all after “#” ignored
  - ❑ reload configuration files using “`-config <file>`”

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 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

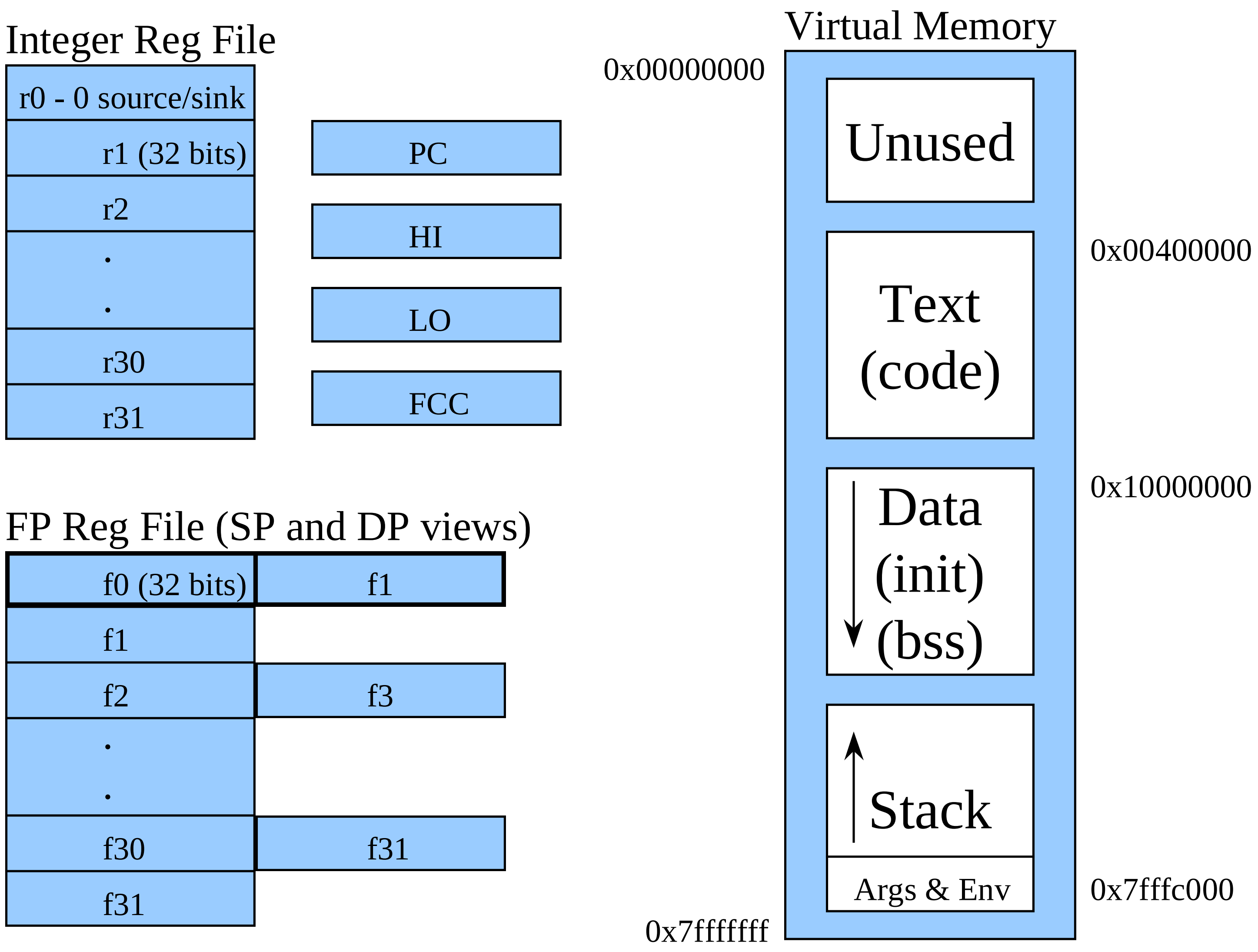
Floating Point Arithmetic:

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

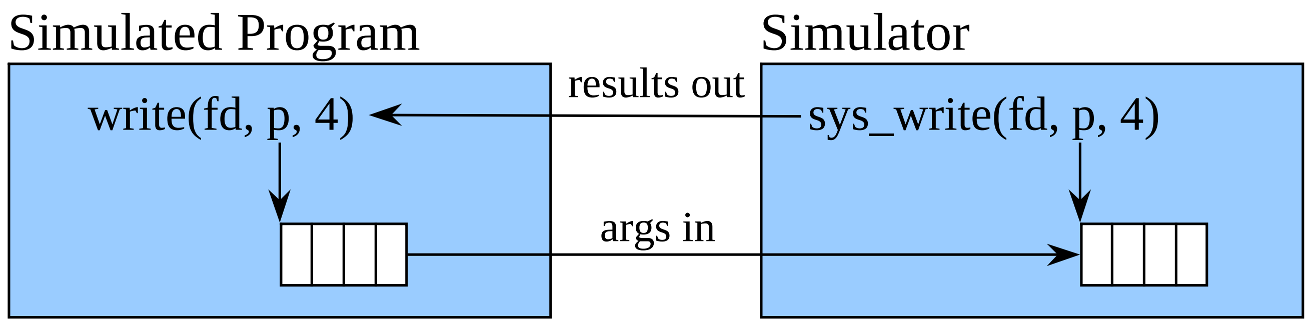
Miscellaneous:

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

SimpleScalar Architected State



Simulator I/O

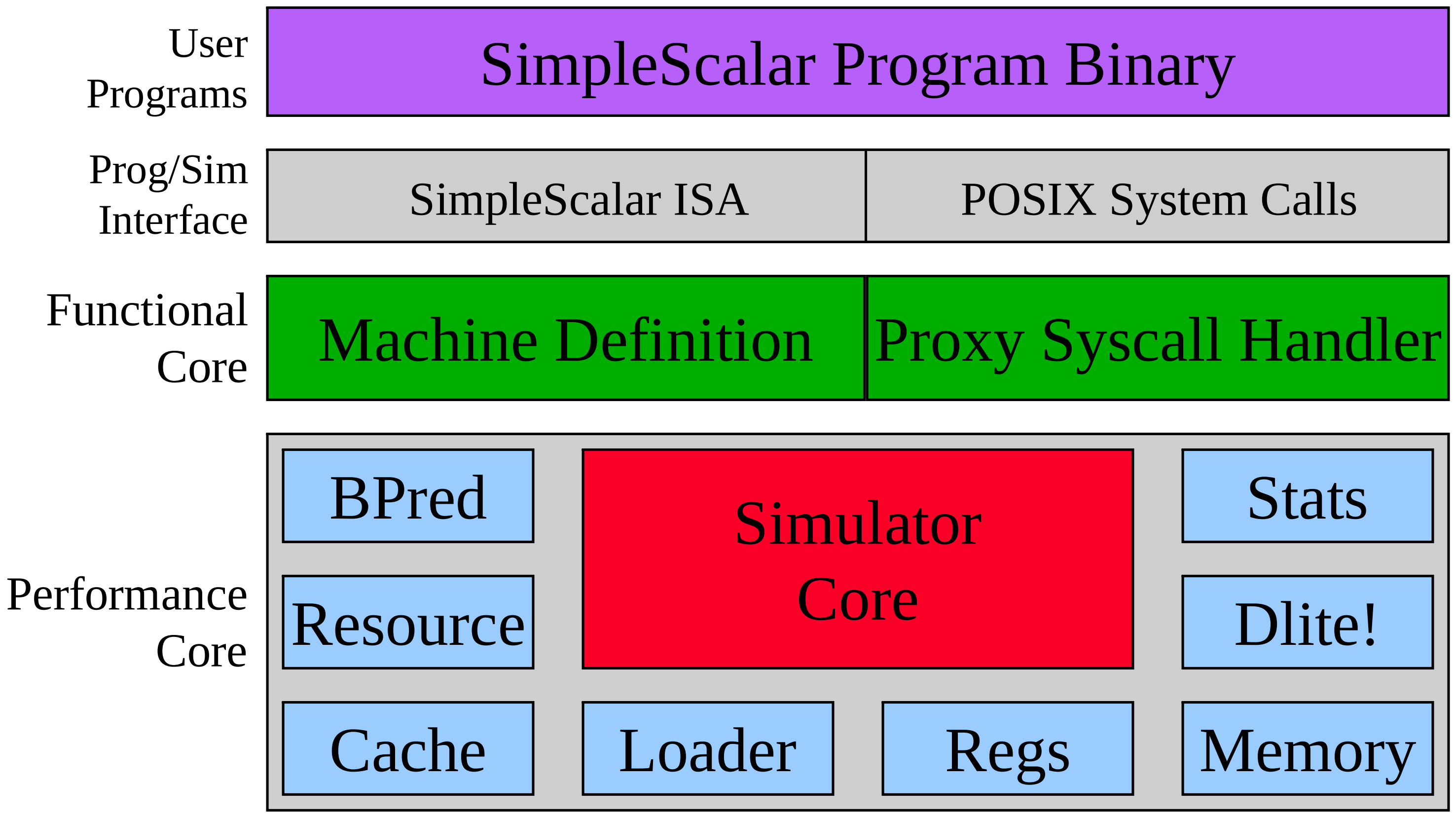


- a useful simulator must implement some form of I/O
  - ❑ I/O implemented via SYSCALL instruction
  - ❑ supports a subset of Ultrix system calls, proxied out to host
- basic algorithm (implemented in syscall.c):
  - ❑ decode system call
  - ❑ copy arguments (if any) into simulator memory
  - ❑ perform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

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

Simulator S/W Architecture



- most of performance core is optional
- most projects will enhance on the “simulator core”

Source Roadmap - Simulator Modules

- sim-safe.c - minimal functional simulator
- sim-fast.c - faster (and twisted) version of sim-safe
- sim-eio.c - EIO trace and checkpoint generator
- sim-profile.c - profiling simulator
- sim-cache.c - two-level cache simulator (no timing)
- sim-cheetah.c - Cheetah single-pass multiple-configuration cache simulator
- sim-bpred.c - branch predictor simulator (no timing)
- sim-outorder.c - detailed OoO issue performance simulator (with timing)

Source Roadmap - System Components

- dlite.[hc] - DLite!, the lightweight debugger
- eio.[hc] - external I/O tracing module
- loader.[hc] - program loader
- memory.[hc] - flat memory space module
- regs.[hc] - register module
- ss.[hc] - SimpleScalar ISA-dependent routines
- ss.def - SimpleScalar ISA definition
- symbol.[hc] - symbol table module
- syscall.[hc] - proxy system call implementation

Source Roadmap - Simulation Components

- bpred.[hc] - branch predictors
- cache.[hc] - cache module
- eventq.[hc] - event queue module
- libcheetah/ - Cheetah cache simulator library
- ptrace.[hc] - pipetrace module
- resources.[hc] - resource manager module
- sim.h - simulator main code interface definitions
- textprof.pl - text segment profile view (Perl Script)
- pipeview.pl - pipetrace view (Perl script)

Source Roadmap - “Really Useful” Modules

- eval.[hc] - generic expression evaluator
- libexo/ - EXO(-skeletal) persistent data structure library
- misc.[hc] - everything miscellaneous
- options.[hc] - options package
- range.[hc] - range expression package
- stats.[hc] - statistics package

Source Roadmap - Build Components

- Makefile - top level make file
- tests/ - standalone self-validating bi-endian test package
- endian.[hc] - endian probes
- main.c - main line code
- sysprobe.c - system probe, used during build process
- version.h - simulator release version information

Source Roadmap - Administrative

- ANNOUNCE - latest release announcement
- CHANGELOG - changes by release
- CONTRIBUTORS - of source, fixes, docs, etc...
- COPYING - SimpleScalar source license
  - free use, all source (C) 1994-1997 by Todd M. Austin
- FAQ - frequently asked questions
  - please read before sending Q's to mailing list or developers
- PROJECTS - various projects we're recruiting for
- README.* - platform-dependent notes
- WARRANTY - none, so don't sue us…
- contrib/ - useful extensions (not yet installed)

Simulator Core Interfaces

```
/* register simulator-specific options */
void sim_reg_options(struct opt_odb_t *odb);

/* check simulator-specific option values */
void sim_check_options(struct opt_odb_t *odb, int argc, char **argv);

/* register simulator-specific statistics */
void sim_reg_stats(struct stat_sdb_t *sdb);

/* initialize the simulator */
void sim_init(void);

/* print simulator-specific configuration information */
void sim_aux_config(FILE *stream);

/* start simulation, program loaded, processor precise state initialized */
void sim_main(void);

/* dump simulator-specific auxiliary simulator statistics */
void sim_aux_stats(FILE *stream);

/* un-initialize simulator-specific state */
void sim_uninit(void);
```

- called in this order (from main.c)
- defined in sim.h

A Minimal Simulator Core (sim-safe.c)

```
/* track number of refs */
static SS_COUNTER_TYPE sim_num_insn = 0;

/* register simulator-specific options */
void
sim_reg_options(struct opt_odb_t *odb)
{
  opt_reg_header(odb,
            "sim-safe: This simulator implements a functional simulator.  This\n"
            "functional simulator is the simplest, most user-friendly simulator in the\n"
            "simplescalar tool set.  Unlike sim-fast, this functional simulator checks\n"
            "for all instruction errors, and the implementation is crafted for clarity\n"
            "rather than speed.\n"
                         );
}

/* check simulator-specific option values */
void
sim_check_options(struct opt_odb_t *odb, int argc, char **argv)
{
}

/* register simulator-specific statistics */
void
sim_reg_stats(struct stat_sdb_t *sdb)
{
  stat_reg_counter(sdb, "sim_num_insn",
                           "total number of instructions executed",
                           &sim_num_insn, sim_num_insn, NULL);
}
```

A Minimal Simulator Core (sim-safe.c)

```
/* initialize the simulator */
void
sim_init(void)
{
  sim_num_refs = 0;

  regs_PC = ld_prog_entry;

  /* pre-decode all instructions (EIO files are pre-pre-decoded) */
  if (sim_eio_fd == NULL)
    {
      SS_ADDR_TYPE addr;

      if (OP_MAX > 255)
            fatal("cannot perform fast decoding, too many opcodes");

      debug("sim: decoding text segment...");
      for (addr=ld_text_base;
               addr < (ld_text_base+ld_text_size);
               addr += SS_INST_SIZE)
            {
              SS_INST_TYPE inst = __UNCHK_MEM_ACCESS(SS_INST_TYPE, addr);
              inst.a = (inst.a & ~0xffff) | (unsigned int)SS_OP_ENUM(SS_OPCODE(inst));
              __UNCHK_MEM_ACCESS(SS_INST_TYPE, addr) = inst;
            }
    }
}
```

A Minimal Simulator Core (sim-safe.c)

```
/* print simulator-specific configuration information */
void
sim_aux_config(FILE *stream)                    /* output stream */
{
}

/* dump simulator-specific auxiliary simulator statistics */
void
sim_aux_stats(FILE *stream)                     /* output stream */
{
}

/* un-initialize simulator-specific state */
void
sim_uninit(void)
{
}
```

A Minimal Simulator Core (sim-safe.c)

```
/*
 * configure the execution engine
 */

/* program counter accessors */
#define SET_NPC(EXPR)                   (next_PC = (EXPR))
#define CPC                             (regs_PC)

/* general purpose registers */
#define GPR(N)                          (regs_R[N])
#define SET_GPR(N,EXPR)                 (regs_R[N] = (EXPR))

/* floating point registers, L->word, F->single-prec, D->double-prec */
#define FPR_L(N)                        (regs_F.l[(N)])
#define SET_FPR_L(N,EXPR)               (regs_F.l[(N)] = (EXPR))
#define FPR_F(N)                        (regs_F.f[(N)])
#define SET_FPR_F(N,EXPR)               (regs_F.f[(N)] = (EXPR))
#define FPR_D(N)                        (regs_F.d[(N) >> 1])
#define SET_FPR_D(N,EXPR)               (regs_F.d[(N) >> 1] = (EXPR))

/* miscellaneous register accessors */
#define SET_HI(EXPR)                    (regs_HI = (EXPR))
#define HI                              (regs_HI)
#define SET_LO(EXPR)                    (regs_LO = (EXPR))
#define LO                              (regs_LO)
#define FCC                             (regs_FCC)
#define SET_FCC(EXPR)                   (regs_FCC = (EXPR))
```

A Minimal Simulator Core (sim-safe.c)

```
/* precise architected memory state helper functions */
#define __READ_WORD(DST_T, SRC_T, SRC)                                                    \
  ((unsigned int)((DST_T)(SRC_T)MEM_READ_WORD(addr = (SRC))))
#define __READ_HALF(DST_T, SRC_T, SRC)                                                    \
  ((unsigned int)((DST_T)(SRC_T)MEM_READ_HALF(addr = (SRC))))
#define __READ_BYTE(DST_T, SRC_T, SRC)                                                    \
  ((unsigned int)((DST_T)(SRC_T)MEM_READ_BYTE(addr = (SRC))))

/* precise architected memory state accessor macros */
#define READ_WORD(SRC)                                                                    \
  __READ_WORD(unsigned int, unsigned int, (SRC))

#define READ_UNSIGNED_HALF(SRC)                                                           \
  __READ_HALF(unsigned int, unsigned short, (SRC))

#define READ_SIGNED_HALF(SRC)                                                             \
  __READ_HALF(signed int, signed short, (SRC))

#define READ_UNSIGNED_BYTE(SRC)                                                           \
  __READ_BYTE(unsigned int, unsigned char, (SRC))

#define READ_SIGNED_BYTE(SRC)                                                             \
  __READ_BYTE(signed int, signed char, (SRC))

#define WRITE_WORD(SRC, DST)                                                              \
  (MEM_WRITE_WORD((DST), (unsigned int)(SRC)))

#define WRITE_HALF(SRC, DST)                                                              \
  (MEM_WRITE_HALF((DST), (unsigned short)(unsigned int)(SRC)))

#define WRITE_BYTE(SRC, DST)                                                              \
  (MEM_WRITE_BYTE((DST), (unsigned char)(unsigned int)(SRC)))
```

A Minimal Simulator Core (sim-safe.c)

```
/* system call handler macro */
#define SYSCALL(INST)                                                       \
  (sim_eio_fd != NULL                                                       \
   ? eio_read_trace(sim_eio_fd, sim_num_insn, mem_access, INST)            \
   : ss_syscall(mem_access, INST))

/* instantiate the helper functions in the '.def' file */
#define DEFINST(OP,MSK,NAME,OPFORM,RES,CLASS,O1,O2,I1,I2,I3,EXPR)
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)
#define CONNECT(OP)
#define IMPL
#include "ss.def"
#undef DEFINST
#undef DEFLINK
#undef CONNECT
#undef IMPL
```

A Minimal Simulator Core (sim-safe.c)

```
/* start simulation, program loaded, processor precise state initialized */
void sim_main(void)
{
  SS_INST_TYPE inst; register SS_ADDR_TYPE next_PC; enum ss_opcode op;

  /* set up initial default next PC */
  next_PC = regs_PC + SS_INST_SIZE;
  while (TRUE)
    {
      /* maintain $r0 semantics */
      regs_R[0] = 0;
      /* keep an instruction count */
      sim_num_insn++;
      /* get the next instruction to execute */
      inst = __UNCHK_MEM_ACCESS(SS_INST_TYPE, regs_PC);

      /* decode the instruction */
      op = SS_OPCODE(inst);
      /* execute instruction */
      switch (op) {
#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"
      default: panic("bogus opcode");
      }

      /* go to the next instruction */
      regs_PC = next_PC; next_PC += SS_INST_SIZE;
    }
}
```

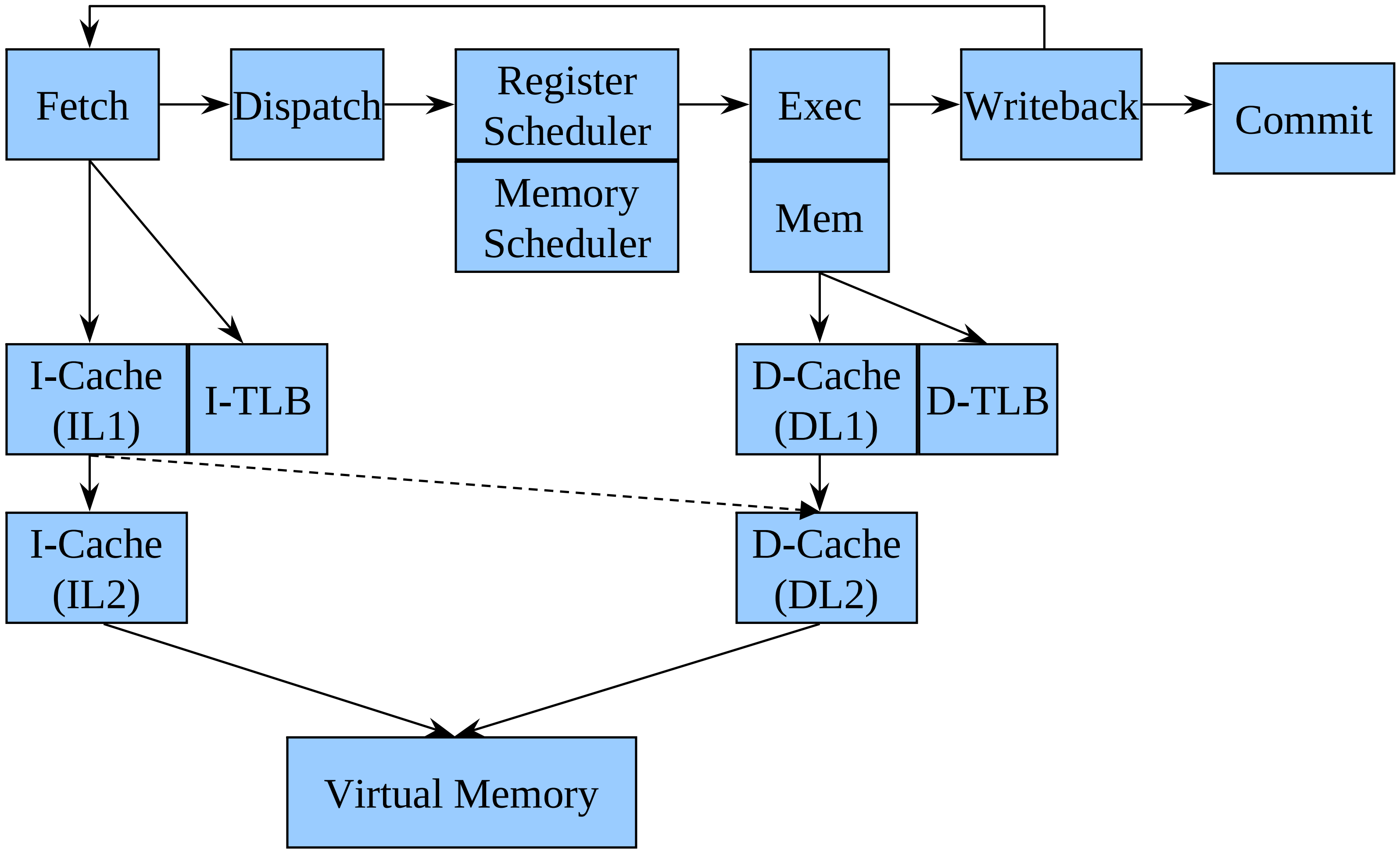
A Minimal Simulator Core (sim-safe.c)

```
#
# Makefile
#

CFLAGS = `./sysprobe -flags` -O0 -g -Wall -DDEBUG
LIBS = `./sysprobe -libs` -lm

SIM_OBJ = main.o syscall.o memory.o regs.o loader.o ss.o endian.o dlite.o \
          symbol.o eval.o options.o stats.o eio.o range.o misc.o \
          libexo/libexo.a

sim-safe:       sysprobe sim-safe.o $(SIM_OBJ)
        $(CC) -o sim-safe $(CFLAGS) sim-safe.o $(SIM_OBJ) $(LIBS)

.c.o:
        $(CC) $(CFLAGS) -c $*.c

sysprobe:       sysprobe.c
        $(CC) $(FFLAGS) -o sysprobe sysprobe.c
```

Tutorial Overview

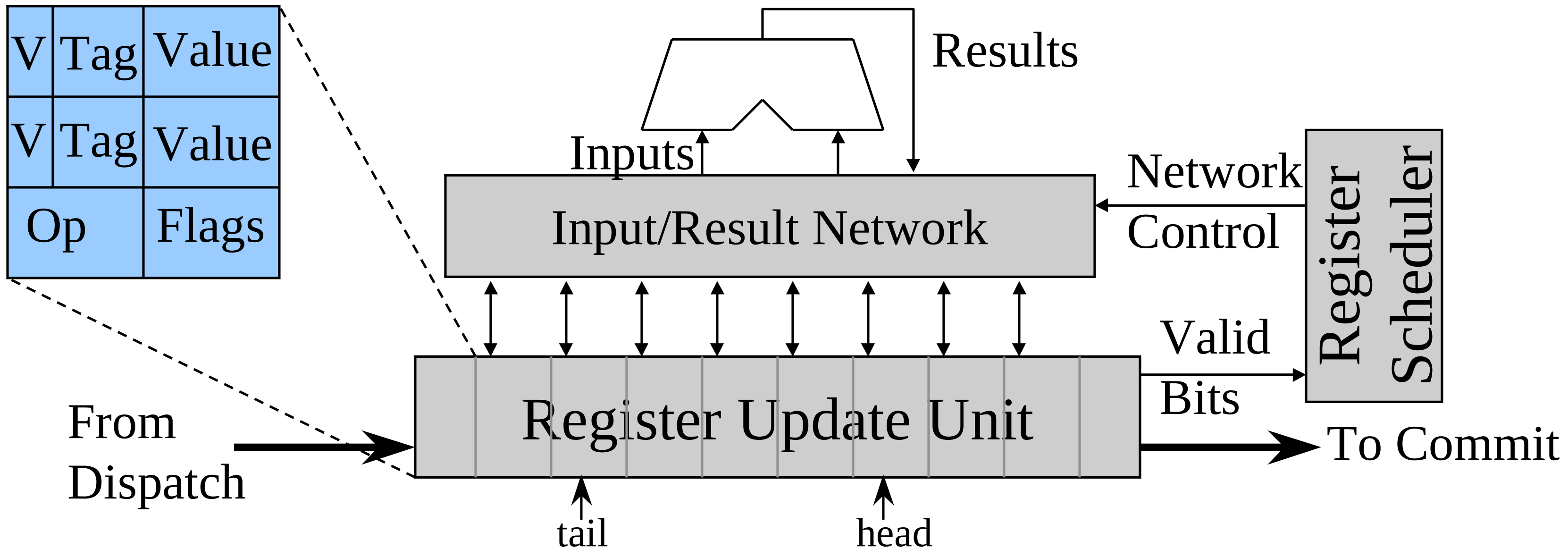
- Overview and Basics
- How to Use the SimpleScalar Architecture and Simulators
- How to Use SIM-OUTORDER
  - ❑ H/W Architecture Overview
  - ❑ S/W Architecture Overview
  - ❑ A Walk Through the Pipeline
  - ❑ Advanced Features
  - ❑ Performance Optimizations
- How to Build Your Own Simulators
- How to Modify the ISA
- ...

SIM-OUTORDER: H/W Architecture



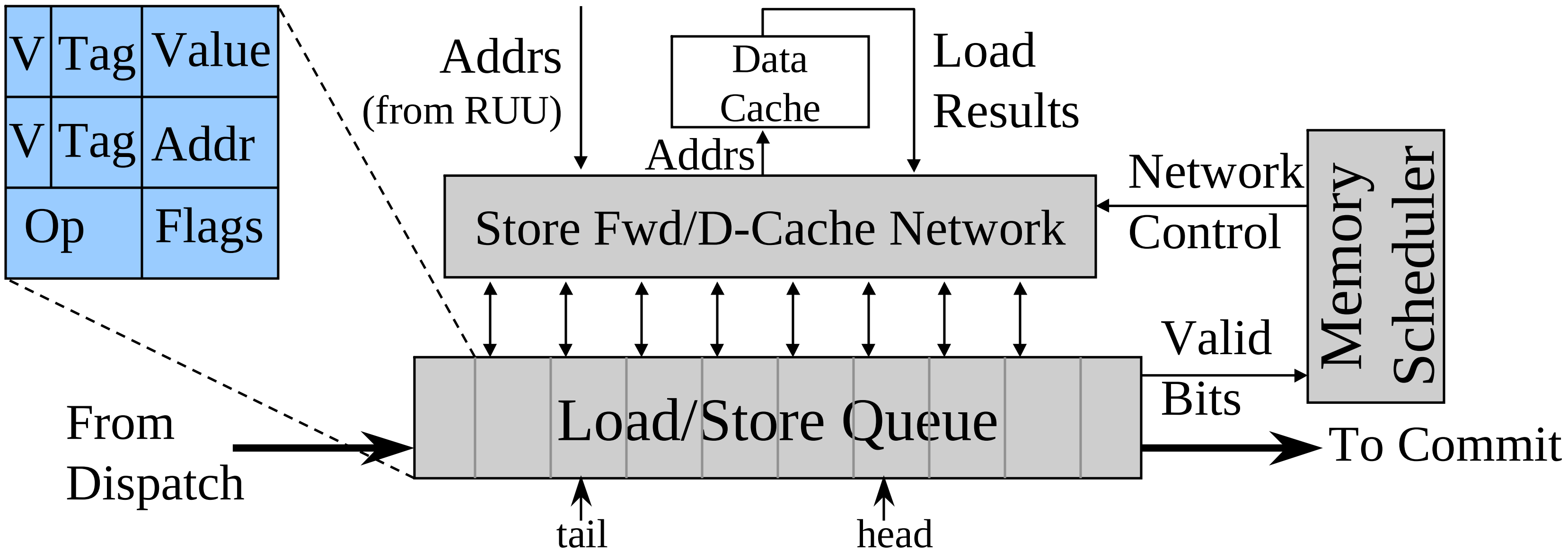
- implemented in `sim-outorder.c` and components

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)



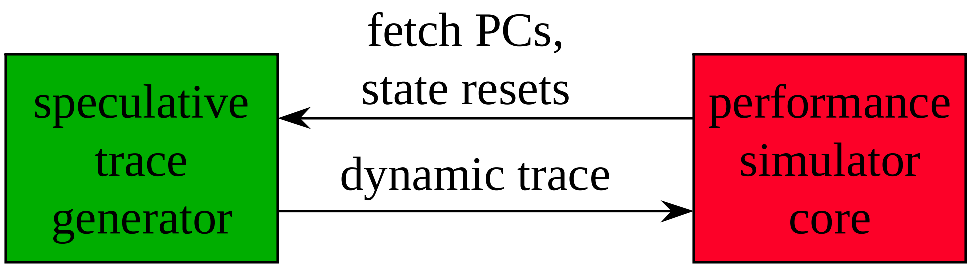
- LSQ handles memory synchronization/communication
  - contains all loads and stores in program order
    - load/store primitives really, address calculation is separate op
    - effective address calculations reside in RUU (as ADD insts)
  - loads issue out-of-order, when memory deps known satisfied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

Main Simulation Loop

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

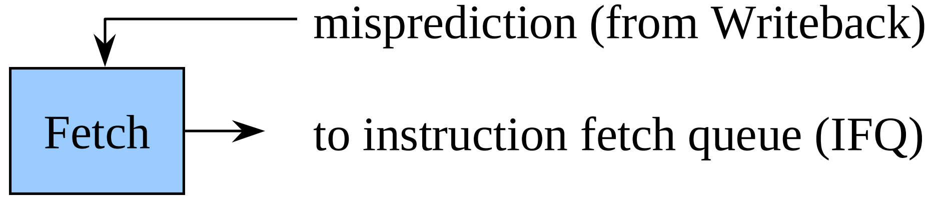
- main simulator loop is implemented in `sim_main()`
- walks pipeline from Commit to Fetch
  - ❑ backward pipeline traversal eliminates relaxation problems, e.g., provides correct inter-stage latch synchronization
- loop is exited via a `longjmp()` to `main()` when simulated program executes an `exit()`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

Speculative Trace Generation



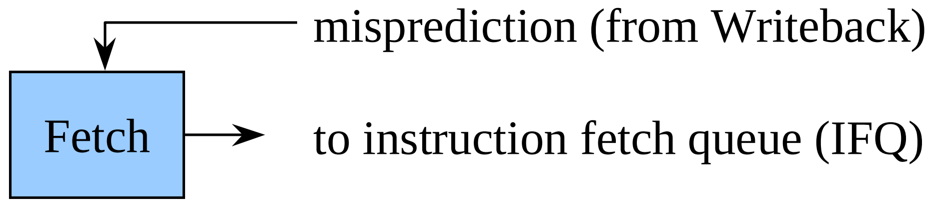
- SIM-OUTORDER is a *dynamic* trace-driven simulator
  - ❑ trace includes correct and misspeculated instructions
  - ❑ tracer controlled by timing model Dispatch and Writeback stages
    - ❑ Dispatch directs execution PCs, Writeback initiates recovery
  - ❑ implemented with same functional component as sim-safe
    - ❑ register/memory macro's redirected to speculative state buffers
    - ❑ committed state written to non-speculative state modules (i.e., memory.c and regs.c)
- permits separation of functional and performance cores
- suffers from imprecise data misspeculation modeling

Fetch Stage Implementation


misprediction (from Writeback)
Fetch
to instruction fetch queue (IFQ)

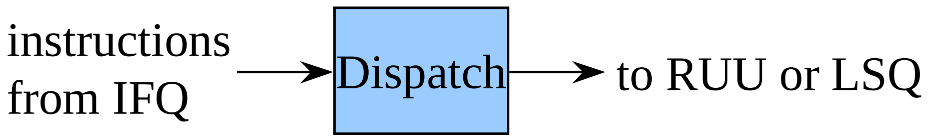
- models machine fetch bandwidth
- implemented in `ruu_fetch()`
- inputs:
  - ❑ program counter
  - ❑ predictor state (see bpred.[hc])
  - ❑ misprediction detection from branch execution unit(s)
- outputs:
  - ❑ fetched instructions sent to instruction fetch queue (IFQ)

Fetch Stage Implementation


misprediction (from Writeback)
Fetch
to instruction fetch queue (IFQ)

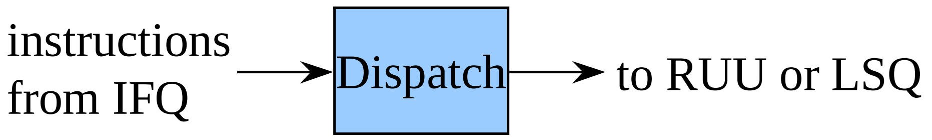
- procedure (once per cycle):
  - fetch instructions from one I-cache line, block until I-cache or I-TLB misses are resolved
  - queue fetched instructions to instruction fetch queue (IFQ)
  - probe branch predictor for cache line to access in next cycle

Dispatch Stage Implementation



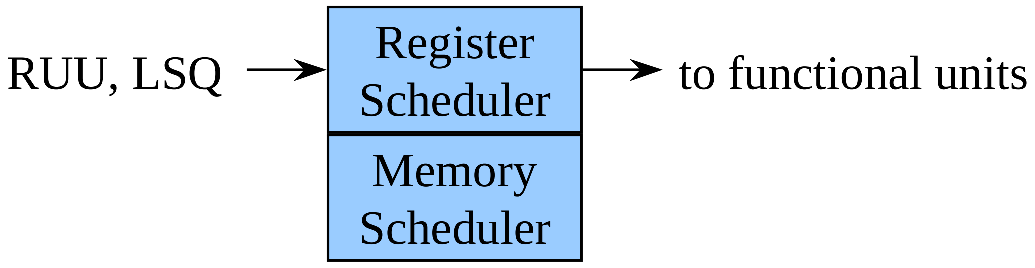
- models machine decode, rename, RUU/LSQ allocation bandwidth, implements register renaming
- implemented in `ruu_dispatch()`
- inputs:
  - instructions from IFQ, from Fetch stage
  - RUU/LSQ occupancy
  - rename table (`create_vector`)
  - architected machine state (for execution)
- outputs:
  - updated RUU/LSQ, rename table, machine state

Dispatch Stage Implementation



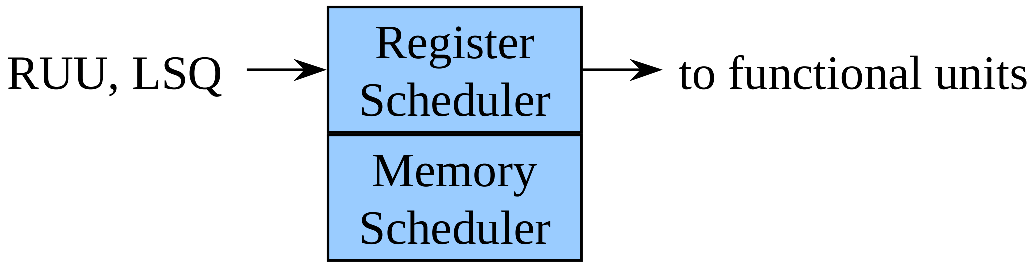
- procedure (once per cycle):
  - fetch insts from IFQ
  - decode and execute instructions
    - permits early detection of branch mis-predicts
    - facilitates simulation of "oracle" studies
  - if branch misprediction occurs:
    - start copy-on-write of architected state to speculative state buffers
  - enter instructions into RUU and LSQ (load/store queue)
    - link to sourcing instruction(s) using RS_LINK structure
    - loads/stores are split into two insts: ADD + Load/Store
      - improves performance of memory dependence checking

Scheduler Stage Implementation



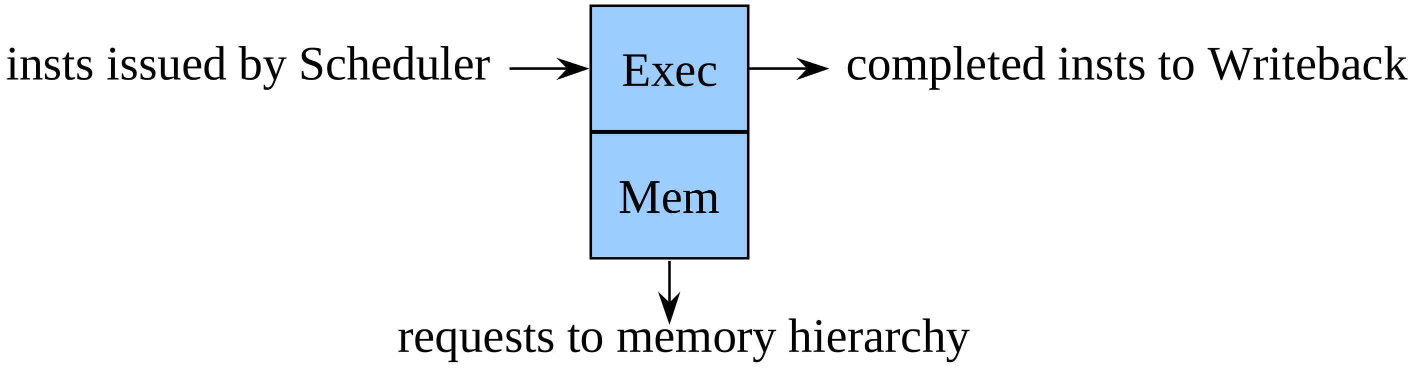
- models instruction wakeup, selection, and issue
  - separate schedulers track register and memory dependencies
- implemented in `ruu_issue()` and `lsq_refresh()`
- inputs:
  - RUU/LSQ
- outputs:
  - updated RUU/LSQ
  - updated functional unit state

Scheduler Stage Implementation



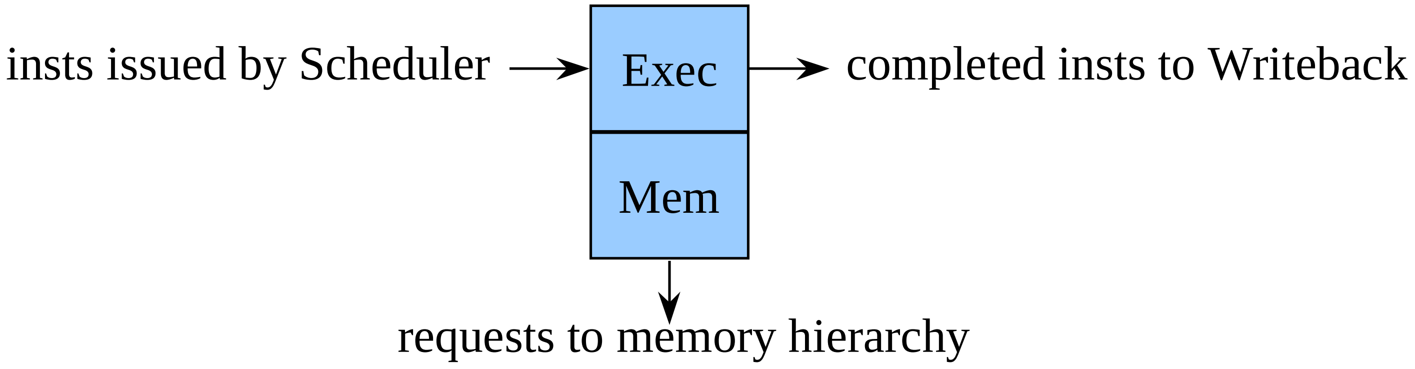
- procedure (once per cycle):
  - locate instructions with all register inputs ready
    - in ready queue, inserted when dependent insts enter Writeback
  - locate loads with all memory inputs ready
    - determined by walking the load/store queue
    - if load addr unknown, then stall issue (and poll again next cycle)
    - if earlier store w/ unknown addr, then stall issue (and poll again)
    - if earlier store w/ matching addr, then forward store data
    - else, access D-cache

Execute Stage Implementation


insts issued by Scheduler
Exec
completed insts to Writeback
Mem
requests to memory hierarchy

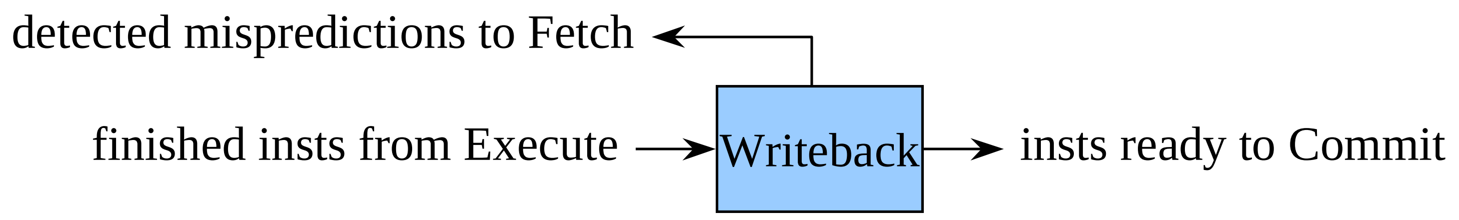
- models functional units and D-cache
  - ❑ access port bandwidths, issue and execute latencies
- implemented in `ruu_issue()`
- inputs:
  - ❑ instructions ready to execute, issued by Scheduler stage
  - ❑ functional unit and D-cache state
- outputs:
  - ❑ updated functional unit and D-cache state, Writeback events

Execute Stage Implementation


insts issued by Scheduler
Exec
completed insts to Writeback
Mem
requests to memory hierarchy

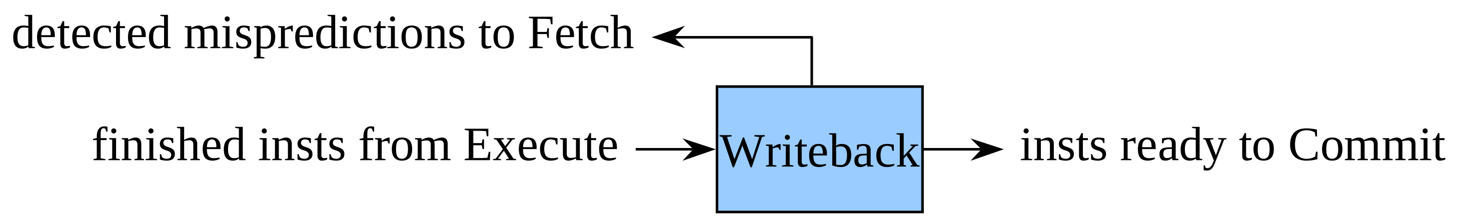
- procedure (once per cycle):
  - get ready instructions (as many as supported by issue B/W)
  - find free functional unit and access port
  - reserve unit for entire issue latency
  - schedule writeback event using operation latency of functional unit
    - for loads satisfied in D-cache, probe D-cache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

Writeback Stage Implementation



- models writeback bandwidth, wakes up ready insts, detects mispredictions, initiated misprediction recovery
- implemented in `ruu_writeback()`
- inputs:
    - ❑ completed instructions as indicated by event queue
    - ❑ RUU/LSQ state (for wakeup walks)
- outputs:
    - ❑ updated event queue, RUU/LSQ, ready queue
    - ❑ branch misprediction recovery updates

Writeback Stage Implementation



- procedure (once per cycle):
  - get finished instructions (specified by event queue)
  - if mispredicted branch, recover state:
    - recover RUU
      - walk newest instruction to mispredicted branch
      - unlink instructions from output dependence chains (tag increment)
    - recover architected state
      - roll back to checkpoint (copy-on-write bits reset, spec mem freed)
  - wakeup walk: walk output dependence chains of finished insts
    - mark dependent instruction's input as now ready
    - if deps satisfied, wake up inst (memory checked in `lsq_refresh()`)

Commit Stage Implementation

insts ready to Commit
Commit

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

Commit Stage Implementation

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

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
```

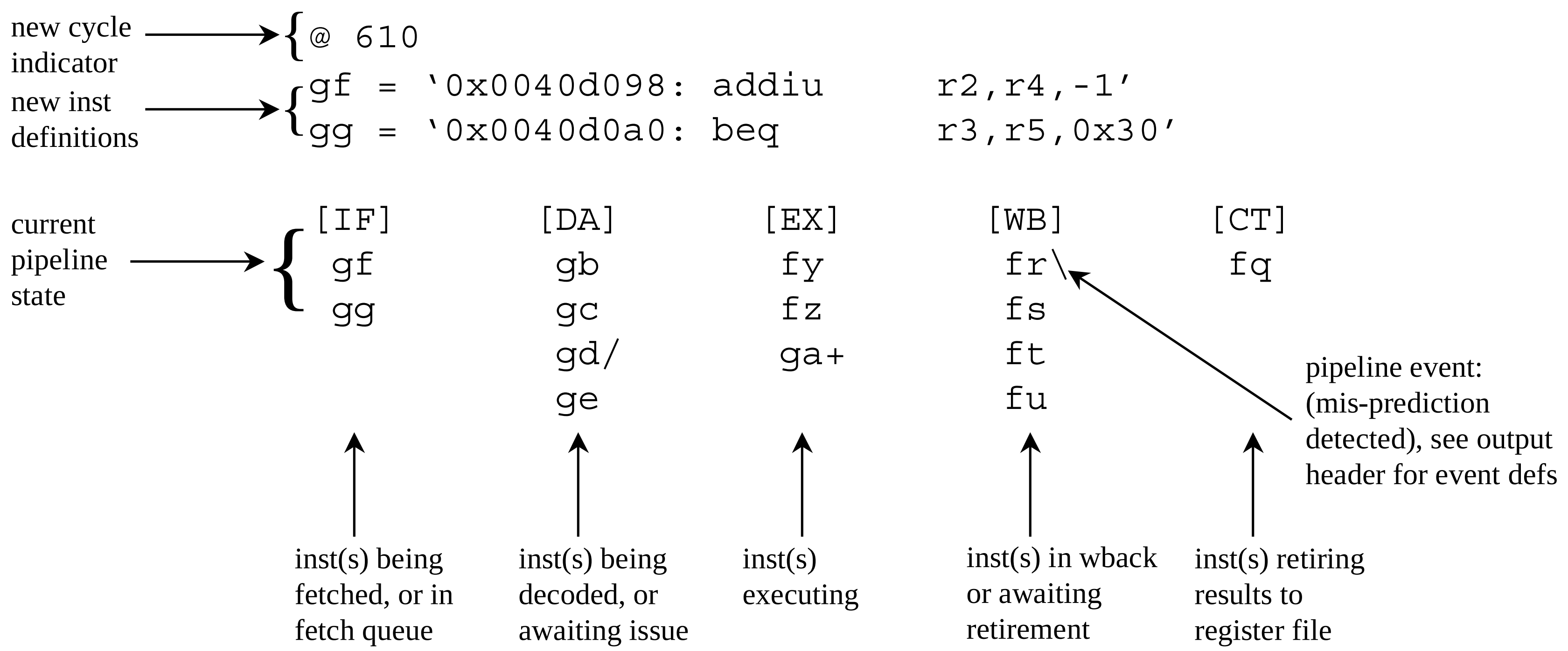
- view with the pipeview.pl Perl script
  - it displays the pipeline for each cycle of execution traced
  - usage: `pipeview.pl <ptrace_file>`

Displaying Pipetraces

- example session:

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

- example output:



```
@ 610
gf = `0x0040d098: addiu     r2,r4,-1'
gg = `0x0040d0a0: beq       r3,r5,0x30'

[IF]      [DA]      [EX]      [WB]      [CT]
 gf        gb        fy        fr\       fq
 gg        gc        fz        fs
           gd/       ga+       ft
           ge                  fu
```

new cycle indicator
new inst definitions
current pipeline state
pipeline event: (mis-prediction detected), see output header for event defs
inst(s) being fetched, or in fetch queue
inst(s) being decoded, or awaiting issue
inst(s) executing
inst(s) in wback or awaiting retirement
inst(s) retiring results to register file

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:

executed 13 times

never executed

```
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!

Optimization: Predecoded Text Segments

```
/* pre-decode all instructions (EIO files are pre-pre-decoded) */
if (sim_eio_fd == NULL)
  {
    SS_ADDR_TYPE addr;

    if (OP_MAX > 255)
          fatal("cannot perform fast decoding, too many opcodes");

    debug("sim: decoding text segment...");
    for (addr=ld_text_base;
         addr < (ld_text_base+ld_text_size);
         addr += SS_INST_SIZE)
          {
            SS_INST_TYPE inst = __UNCHK_MEM_ACCESS(SS_INST_TYPE, addr);
            inst.a = (inst.a & ~0xffff) | (unsigned int)SS_OP_ENUM(SS_OPCODE(inst));
            __UNCHK_MEM_ACCESS(SS_INST_TYPE, addr) = inst;
          }
  }
```

- instruction opcodes replaced with internal enum
  - ❑ speeds decode, by eliminating it...
  - ❑ note: EIO text segments are pre-pre-decoded
- leverages pristine SimpleScalar binary organization
  - ❑ all 8-byte entries in text segment are insts or unused space

Optimization: Output Dependence Chains

```
/* a reservation station link: this structure links elements of a RUU
   reservation station list; used for ready instruction queue, event queue, and
   output dependency lists; each RS_LINK node contains a pointer to the RUU
   entry it references along with an instance tag, the RS_LINK is only valid if
   the instruction instance tag matches the instruction RUU entry instance tag;
   this strategy allows entries in the RUU can be squashed and reused without
   updating the lists that point to it, which significantly improves the
   performance of (all to frequent) squash events */
struct RS_link {
  struct RS_link *next;                /* next entry in list */
  struct RUU_station *rs;              /* referenced RUU resv station */
  INST_TAG_TYPE tag;                   /* inst instance sequence number */
  union {
    SS_TIME_TYPE when;                 /* time stamp of entry (for eventq) */
    INST_SEQ_TYPE seq;                 /* inst sequence */
    int opnum;                         /* input/output operand number */
  } x;
};
```

- register dependencies described with dependence chains
  - ❑ rooted in RUU of defining instruction, one per output register
  - ❑ also rooted in create vector, at index of logical register
- output dependence chains walked during Writeback
  - same links used for event queue, ready queue, etc...

Optimization: Output Dependence Chains

```
/* link RS onto the output chain number of whichever operation will create reg */
static INLINE void ruu_link_idep(struct RUU_station *rs, int idep_num, int idep_name)
{
  struct CV_link head; struct RS_link *link;

  /* any dependence? */
  if (idep_name == NA) {
      /* no input dependence for this input slot, mark operand as ready */
      rs->idep_ready[idep_num] = TRUE;
      return;
    }

  /* locate creator of operand */
  head = CREATE_VECTOR(idep_name);

  /* any creator? */
  if (!head.rs) {
      /* no active creator, use value available in architected reg file,
         indicate the operand is ready for use */
      rs->idep_ready[idep_num] = TRUE;
      return;
    }
  /* else, creator operation will make this value sometime in the future */

  /* indicate value will be created sometime in the future, i.e., operand
     is not yet ready for use */
  rs->idep_ready[idep_num] = FALSE;

  /* link onto creator’s output list of dependant operand */
  RSLINK_NEW(link, rs); link->x.opnum = idep_num;
  link->next = head.rs->odep_list[head.odep_num];
  head.rs->odep_list[head.odep_num] = link;
}
```

Optimization: Tagged Dependence Chains

- observation: “squash” recovery consumes many cycles
  - leverage “tagged” pointers to speed squash recover
  - unique tag assigned to each instruction, copied into references
  - squash an entry by destroying tag, makes all references stale

```
/* in ruu_recover(): squash this RUU entry */
RUU[RUU_index].tag++;
```

- all dereferences must check for stale references

```
/* walk output list, queue up ready operations */
for (olink=rs->odep_list[i]; olink; olink=olink_next)
  {
    if (RSLINK_VALID(olink)) {
        /* input is now ready */
        olink->rs->idep_ready[olink->x.opnum] = TRUE;
      }
    . . .

    /* grab link to next element prior to free */
    olink_next = olink->next;
  }
```

Optimization: Fast Functional State Recovery

```
/* speculation mode, non-zero when mis-speculating */
static int spec_mode = FALSE;

/* integer register file */
static BITMAP_TYPE(SS_NUM_REGS, use_spec_R);
static SS_WORD_TYPE spec_regs_R[SS_NUM_REGS];

/* general purpose register accessors */
#define GPR(N)                    (BITMAP_SET_P(use_spec_R, R_BMAP_SZ, (N))\
                                   ? spec_regs_R[N]                       \
                                   : regs_R[N])
#define SET_GPR(N,EXPR)           (spec_mode                              \
                                   ? (spec_regs_R[N] = (EXPR),            \
                                      BITMAP_SET(use_spec_R, R_BMAP_SZ, (N)),\
                                      spec_regs_R[N])                     \
                                   : (regs_R[N] = (EXPR)))

/* reset copied-on-write register bitmasks back to non-speculative state */
BITMAP_CLEAR_MAP(use_spec_R, R_BMAP_SZ);

/* speculative memory hash table */
static struct spec_mem_ent *store_htable[STORE_HASH_SIZE];
```

- early execution permits early detection of mispeculation
  - ❑ when misspeculation begins, all new state definitions redirected
  - ❑ copy-on-write bits indicate speculative defs, reset on recovery
  - ❑ speculative memory defs in store hash table, flushed on recovery

Tutorial Overview

- Overview and Basics
- How to Use the SimpleScalar Architecture and Simulators
- How to Use SIM-OUTORDER
- How to Build Your Own Simulators
  - ❑ Overview
  - ❑ “Really Useful” Modules
  - ❑ Architecture-related Modules
  - ❑ Simulation Modules
- How to Modify the ISA
- ...

Source Roadmap - “Really Useful” Modules

- eval.[hc] - generic expression evaluator
- libexo/ - EXO(-skeletal) persistent data structure library
- misc.[hc] - everything miscellaneous
- options.[hc] - options package
- range.[hc] - range expression package
- stats.[hc] - statistics package

Options Package (option.[hc])

```
/* create a new option database */
struct opt_odb_t *opt_new(orphan_fn_t orphan_fn);

/* free an option database */
void opt_delete(struct opt_odb_t *odb);

/* register an integer option variable */
void opt_reg_int(struct opt_odb_t *odb,      /* option database */
                 char *name,                 /* option name */
                 char *desc,                 /* option description */
                 int *var,                   /* pointer to option variable */
                 int def_val,                /* default value of option variable */
                 int print);                 /* print during '-dumpconfig' */

/* process command line arguments */
void opt_process_options(struct opt_odb_t *odb, int argc, char **argv);

/* print all options and current values */
void opt_print_options(struct opt_odb_t *odb, FILE *fd, int terse, int notes);

/* print option help page with default values */
void opt_print_help(struct opt_odb_t *odb, FILE *fd);
```

- option variables are registered (by type) into option DB
  - ❑ integer, float, enum, boolean types supported (plus lists)
  - ❑ builtin support to save/restore options (`-dumpconfig, -config`)
- program headers and option notes also supported

Statistics Package (stats.[hc])

```
/* create, delete stats databases */
struct stat_sdb_t *stat_new(void);
void stat_delete(struct stat_sdb_t *sdb);

/* register an integer statistical variable */
struct stat_stat_t *
stat_reg_int(struct stat_sdb_t *sdb,    /* stat database */
             char *name,                /* stat variable name */
             char *desc,                /* stat variable description */
             int *var,                  /* stat variable */
             int init_val);              /* stat variable initial value */

/* register a statistical formula */
struct stat_stat_t *stat_reg_formula(struct stat_sdb_t *sdb,/* stat database */
                                     char *name,            /* stat variable name */
                                     char *desc,            /* stat variable description */
                                     char *formula);         /* formula expression */

/* create an array distributions, array and sparse arrays */
struct stat_stat_t *stat_reg_dist(...);
struct stat_stat_t *stat_reg_sdist(...);

/* print the value of all stat variables in stat database SDB */
void stat_print_stats(struct stat_sdb_t *sdb, FILE *fd);
```

- provides counters, expressions, and distributions
  - ❑ register integers, floats, counters, create distributions
- manipulate stats counters directly, e.g., `stat_num_insn++`

Miscellaneous Functions (misc.[hc])

```
/* declare a fatal run-time error, calls fatal hook function */
void fatal(char *fmt, ...);

/* declare a panic situation, dumps core */
void panic(char *fmt, ...);

/* declare a warning */
void warn(char *fmt, ...);

/* print general information */
void info(char *fmt, ...);

/* print a debugging message */
void debug(char *fmt, ...);

/* return string describing elapsed time, passed in SEC in seconds */
char *elapsed_time(long sec);

/* allocate some core, this memory has overhead no larger than a page
   in size and it cannot be released. the storage is returned cleared */
void *getcore(int nbytes);
```

- lots of useful stuff in here
  - ❑ more features when compiled with GCC
- many portability issues resolved in this module
  - ❑ e.g., `mystrdup()`, `mystrrcmp()`, etc...

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

DLite!, the Lite Debugger (dlite.[hc])

```
/* initialize the DLite debugger */
void
dlite_init(dlite_reg_obj_t reg_obj,            /* register state object */
           dlite_mem_obj_t mem_obj,            /* memory state object */
           dlite_mstate_obj_t mstate_obj);     /* machine state object */

/* check for a break condition */
#define dlite_check_break(NPC, ACCESS, ADDR, ICNT, CYCLE)              \
  ((dlite_check || dlite_active)                                       \
   ? __check_break((NPC), (ACCESS), (ADDR), (ICNT), (CYCLE))           \
   : FALSE)

/* DLite debugger main loop */
void
dlite_main(SS_ADDR_TYPE regs_PC,               /* addr of last inst to exec */
           SS_ADDR_TYPE next_PC,               /* addr of next inst to exec */
           SS_COUNTER_TYPE cycle);             /* current cycle */
```

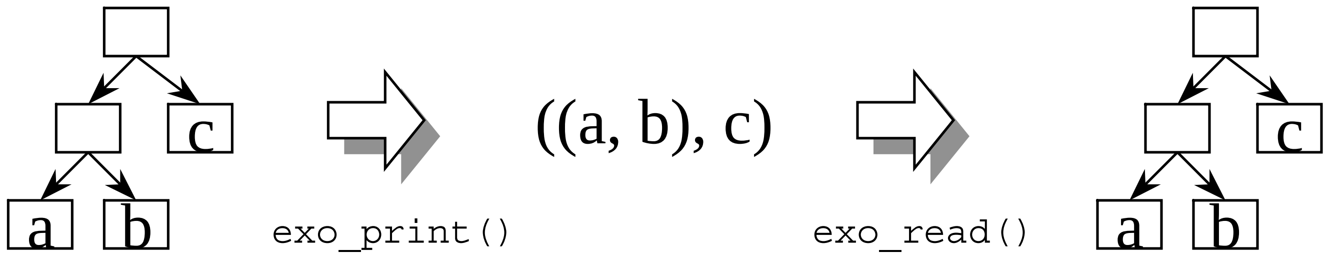
- initialize debugger with state accessor functions
- call check interface each cycle with indication of upcoming execution events
- call main line debugger interface when check function requests control

Symbol Table Module (symbol.[hc])

```
/* internal program symbol format */
struct sym_sym_t {
  char *name;                    /* symbol name */
  enum sym_seg_t seg;            /* symbol segment */
  int initialized;               /* initialized? (if data segment) */
  int pub;                       /* externally visible? */
  int local;                     /* compiler local symbol? */
  SS_ADDR_TYPE addr;             /* symbol address value */
  int size;                      /* bytes to next symbol */
};

/* bind address ADDR to a symbol in symbol database DB */
struct sym_sym_t *                       /* symbol found, or NULL */
sym_bind_addr(SS_ADDR_TYPE addr,         /* address of symbol to locate */
              int *pindex,               /* ptr to index result var */
              int exact,                 /* require exact address match? */
              enum sym_db_t db);         /* symbol database to search */

/* bind name NAME to a symbol in symbol database DB */
struct sym_sym_t *                               /* symbol found, or NULL */
sym_bind_name(char *name,                        /* symbol name to locate */
              int *pindex,                       /* ptr to index result var */
              enum sym_db_t db);                 /* symbol database to search */
```

- symbols loaded at startup
- sorted tables also available, see `symbol.h`

EXO Library (libexo/)



```
/* create a new EXO term */
struct exo_term_t *exo_new(ec_integer, (exo_integer_t)<int>);
struct exo_term_t *exo_new(ec_string, "<string>");
struct exo_term_t *exo_new(ec_list, <list_ent>..., NULL);
struct exo_term_t *exo_new(ec_array, <size>, <array_ent>..., NULL);

/* release an EXO term */
void exo_delete(struct exo_term_t *exo);

/* print/read an EXO term */
void exo_print(struct exo_term_t *exo, FILE *stream);
struct exo_term_t *exo_read(FILE *stream);
```

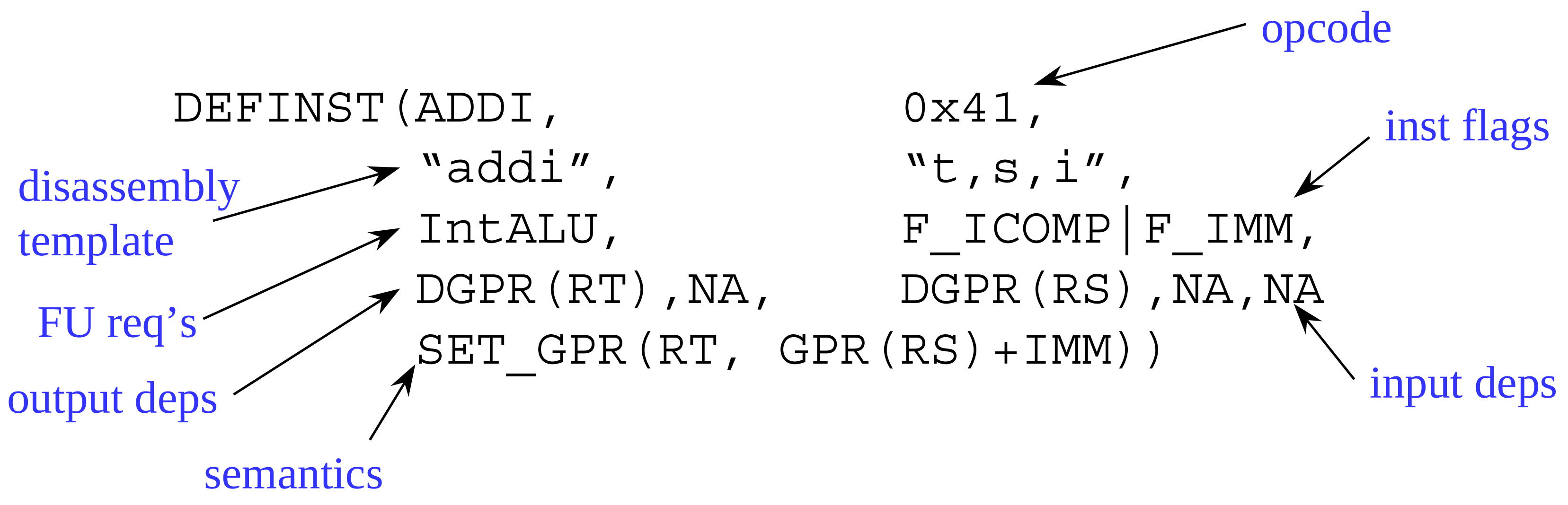
- for saving and restoring data structures to files
  - ❑ convert structure to EXO format, print to stream, read later
- use by EIO traces, useful for saving/restoring profiles

Source Roadmap - System Components

- dlite.[hc] - DLite!, the lightweight debugger
- eio.[hc] - external I/O tracing module
- loader.[hc] - program loader
- memory.[hc] - flat memory space module
- regs.[hc] - register module
- ss.[hc] - SimpleScalar ISA-dependent routines
- ss.def - SimpleScalar ISA definition
- symbol.[hc] - symbol table module
- syscall.[hc] - proxy system call implementation

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:



```
DEFINST(ADDI,                0x41,
        "addi",              "t,s,i",
        IntALU,              F_ICOMP|F_IMM,
        DGPR(RT),NA,         DGPR(RS),NA,NA
        SET_GPR(RT, GPR(RS)+IMM))
```

Input/Output Dependency Specification

```
DGPR(N)   - general purpose register N
DGPR_D(N) - double word general purpose register N

DFPR_L(N) - floating-point register N, as word
DFPR_F(N) - floating-point reg N, as single-prec float
DFPR_D(N) - floating-point reg N, as double-prec double

DHI       - HI result register
DLO       - LO result register
DFCC      - floating point condition codes
DCPC      - current PC
DNPC      - next PC
DNA       - no dependence
```

- for each inst, dependencies described with macro list:
  - two output dependencies
  - three input dependencies
- examples uses:
  - define macros to produce rename index mapping
  - define macros to access input and output values

Crafting an Dependence Decoder

```
#define DGPR(N)                    (N)
#define DFPR_F(N)                  (32+(N))
…etc…

switch (SS_OPCODE(inst)) {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,CLASS,O1,O2,I1,I2,I3,EXPR)   \
      case OP:                                                     \
        out1 = O1; out2 = O2;                                      \
        in1 = I1; in2 = I2; in3 = 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;
    }
```

Instruction Semantics Specification

```
GPR(N)          - read general purpose register N
SET_GPR(N,E)    - write general purpose register N with E
GPR_D(N)        - read double word general purpose reg N
SET_GPR_D(N,E) - write double word gen purpose reg N w/ E
FPR_L(N)        - read floating-point register N, as word
SET_FPR_L(N,E) - floating-point reg N, as word, with E
FPR_F(N)        - read FP reg N, as single-prec float
SET_FPR_F(N,E) - write FP reg N, as single-prec float w/ E
FPR_D(N)        - read FP reg N, as double-prec double
SET_FPR_D(N,E) - write FP reg N, as double-prec double w/E
HI              - read HI result register
SET_HI(E)       - write HI result register with E
LO              - read LO result register
SET_LO(E)       - write LO result register with E
FCC             - read floating point condition codes
SET_FCC(E)      - write floating point condition codes w/ E
CPC             - read current PC register
NPC             - read next PC register
SET_NPC(E)      - write next PC register with E
TPC             - read target PC register
SET_TPC(E)      - write target PC register with E

READ_SIGNED_BYTE(A)    - read signed byte from address A
READ_UNSIGNED_BYTE(A) - read unsigned byte from address A
READ_SIGNED_HALF(A)    - read signed half from address A
READ_UNSIGNED_HALF(A) - read unsigned half from address A
READ_WORD(A)           - read word from address A
WRITE_BYTE(E,A)        - write byte value E to address A
WRITE_HALF(E,A)        - write half value E to address A
WRITE_WORD(E,A)        - write word value E to address A
```

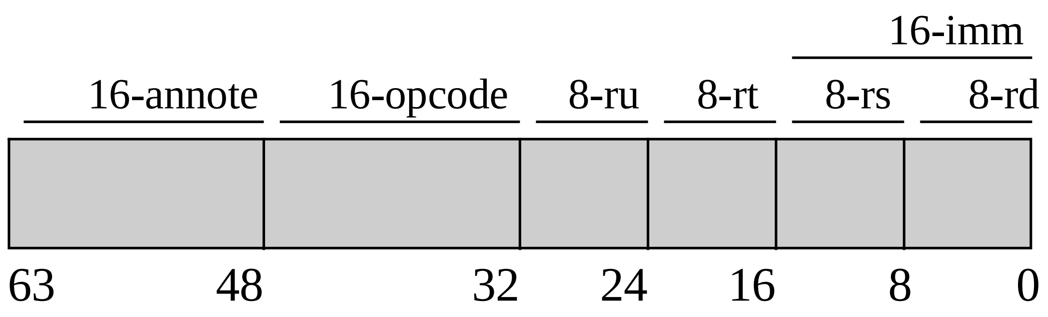
- define as per state implementation of simulator

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))
…etc…


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
    }
```

Instruction Field Accessors

```
RS      - RS register field value
RT      - RT register field value
RD      - RD register field value
FS      - RS register field value
FT      - RT register field value
FD      - RD register field value
BS      - RS register field value
TARG    - jump target field value
OFS     - signed offset field value
IMM     - signed offset field value
UIMM    - unsigned offset field value
SHAMT   - shift amount field value
BCODE   - break code field value
```



- instruction assumed to be in variable `inst`

SimpleScalar ISA Module (ss.[hc])

```
/* returns the opcode field value of SimpleScalar instruction INST */
#define SS_OPCODE(INST)         (INST.a & 0xff)

/* inst opcode -> enum ss_opcode mapping, use this macro to decode insts */
#define SS_OP_ENUM(MSK)         (ss_mask2op[MSK])

/* enum ss_opcode -> opcode operand format, used by disassembler */
#define SS_OP_FORMAT(OP)        (ss_op2format[OP])
extern char *ss_op2format[];

/* enum ss_opcode -> enum ss_fu_class, used by performance simulators */
#define SS_OP_FUCLASS(OP)       (ss_op2fu[OP])

/* enum ss_opcode -> opcode flags, used by simulators */
#define SS_OP_FLAGS(OP)         (ss_op2flags[OP])

/* disassemble a SimpleScalar instruction */
void
ss_print_insn(SS_INST_TYPE inst,        /* instruction to disassemble */
              SS_ADDR_TYPE pc,          /* addr of inst, used for PC-rels */
              FILE *stream);            /* output stream */
```

- add flags to ss.def file as per project requirements:
  - ❑ define `F_xxx` flag in ss.h
  - ❑ probe the new flag with `SS_OP_FLAGS()` macro
  - ❑ e.g., `if (SS_OP_FLAGS(opcode) & (F_MEM|F_LOAD)) == (F_MEM|F_LOAD))`

Register Module (reg.[hc])

```
/* (signed) integer register file */
extern SS_WORD_TYPE regs_R[SS_NUM_REGS];

/* floating point register file format */
extern union regs_FP {
    SS_WORD_TYPE l[SS_NUM_REGS];                  /* integer word view */
    SS_FLOAT_TYPE f[SS_NUM_REGS];                 /* single-precision FP view */
    SS_DOUBLE_TYPE d[SS_NUM_REGS/2];              /* double-precision FP view */
} regs_F;

/* miscellaneous registers */
extern SS_WORD_TYPE regs_HI, regs_LO;
extern int regs_FCC;
extern SS_ADDR_TYPE regs_PC;

/* initialize register file */
void regs_init(void);
```

- access non-speculative registers directly:
  - ❑ e.g., `regs_R[5] = 12;`
- floating point register file supports three “views”:
  - ❑ integers (used by loads), single-precision, double-precision
  - ❑ e.g., `regs_F.f[4] = 23.5;`

Memory Module (memory.[hc])

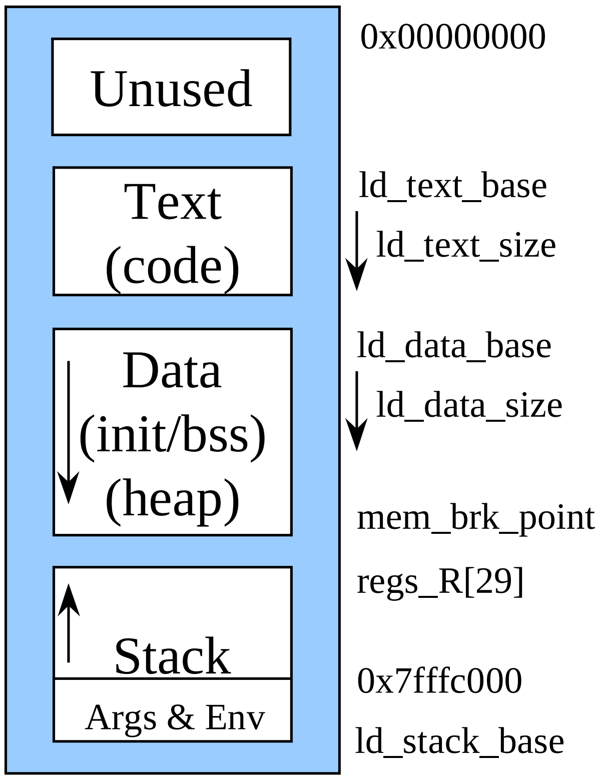
```
/* determines if the memory access is valid, returns error string or NULL */
char *mem_valid(enum mem_cmd cmd,       /* Read (from sim mem) or Write */
                SS_ADDR_TYPE addr,      /* target address to access */
                int nbytes,             /* number of bytes to access */
                int declare);           /* declare any detected error? */

/* generic memory access function, its safe… */
void mem_access(enum mem_cmd cmd,       /* Read (from sim mem) or Write */
             SS_ADDR_TYPE addr,         /* target address to access */
             void *vp,                  /* host memory address to access */
             int nbytes);               /* number of bytes to access */

/* memory access macros, these are safe */
#define MEM_READ_WORD(addr)             …
#define MEM_WRITE_WORD(addr, word)      …
#define MEM_READ_HALF(addr)             …
#define MEM_WRITE_HALF(addr, half)      …
#define MEM_READ_BYTE(addr)             …
#define MEM_WRITE_BYTE(addr, byte)      …
```

- implements large flat memory spaces
  - ❑ may be used to implement virtual or physical memory
  - ❑ uses single-level page table implementation
- safe and unsafe version of all interfaces

Loader Module (loader.[hc])

```
/* load program text and initialized data into simulated virtual memory
   space and initialize program segment range variables */
void ld_load_prog(mem_access_fn mem_fn, /* user-specified memory accessor */
                  int argc, char **argv,/* simulated program cmd line args */
                  char **envp,          /* simulated program environment */
                  int zero_bss_segs);   /* zero uninit data segment? */
```

- prepares program memory for execution
  - ❑ loads program text section (code)
  - ❑ loads program data sections
  - ❑ initializes BSS section
  - ❑ sets up initial call stack
    - ❑ program arguments (argv)
    - ❑ user environment (envp)



Source Roadmap - Simulation Components

- bpred.[hc] - branch predictors
- cache.[hc] - cache module
- eventq.[hc] - event queue module
- libcheetah/ - Cheetah cache simulator library
- ptrace.[hc] - pipetrace module
- resources.[hc] - resource manager module
- sim.h - simulator main code interface definitions
- textprof.pl - text segment profile view (Perl Script)
- pipeview.pl - pipetrace view (Perl script)

Resource Manager (resource.[hc])

```
/* resource descriptor */
struct res_desc {
  char *name;                         /* name of functional unit */
  int quantity;                       /* total instances of this unit */
  int busy;                           /* non-zero if this unit is busy */
  struct res_template {
    int class;                        /* matching resource class */
    int oplat;                        /* operation latency */
    int issuelat;                     /* issue latency */
  } x[MAX_RES_CLASSES];
};

/* create a resource pool */
struct res_pool *res_create_pool(char *name, struct res_desc *pool, int ndesc);

/* get a free resource from resource pool POOL */
struct res_template *res_get(struct res_pool *pool, int class);
```

- generic resource manager
  - ❑ handles most any resource, e.g., ports, fn units, buses, etc...
  - ❑ manager maintains resource availability
  - ❑ configure with a resource descriptor list
  - ❑ `busy` = cycles until available

Resource Manager (resource.[hc])

```
/* resource pool configuration */
struct res_desc fu_config[] = {
  { "integer-ALU", 4, 0,
    { { IntALU, 1, 1 } } },
  { "integer-MULT/DIV", 1, 0,
    { { IntMULT, 3, 1 }, { IntDIV, 20, 19 } } },
  { "memory-port", 2, 0,
    { { RdPort, 1, 1 }, { WrPort, 1, 1 } } }
};
```

- resource pool configuration:
  - instantiate with configuration descriptor list
    - i.e., `{ "name", num, { FU_class, issue_lat, op_lat }, … }`
  - one entry per "type" of resource
  - class IDs indicate services provided by resource instance
  - multiple resource "types" can service same class ID
    - earlier entries in list given higher priority

Branch Predictors (bpred.[hc])

```
/* create a branch predictor */
struct bpred *                         /* branch predictory instance */
bpred_create(enum bpred_class class,   /* type of predictor to create */
             unsigned int bimod_size,  /* bimod table size */
             unsigned int l1size,      /* level-1 table size */
             unsigned int l2size,      /* level-2 table size */
             unsigned int meta_size,   /* meta predictor table size */
             unsigned int shift_width, /* history register width */
             unsigned int xor,         /* history xor address flag */
             unsigned int btb_sets,    /* number of sets in BTB */
             unsigned int btb_assoc,   /* BTB associativity */
             unsigned int retstack_size);/* num entries in ret-addr stack */

/* register branch predictor stats */
void
bpred_reg_stats(struct bpred *pred,    /* branch predictor instance */
                struct stat_sdb_t *sdb);/* stats database */
```

- various branch predictors implemented:
  - ❑ direction: static, bimodal, 2-level adaptive, global, gshare, hybrid
  - ❑ address: return, BTB
- call `bpred_reg_stats` to register predictor-dependent stats

Branch Predictors (bpred.[hc])

```
/* probe a predictor for a next fetch address */
SS_ADDR_TYPE                            /* predicted branch target addr */
bpred_lookup(struct bpred *pred,        /* branch predictor instance */
             SS_ADDR_TYPE baddr,        /* branch address */
             SS_ADDR_TYPE btarget,      /* branch target if taken */
             enum ss_opcode op,         /* opcode of instruction */
             int r31p,                  /* is this using r31? */
             struct bpred_update *dir_update_ptr,/* predictor state pointer */
             int *stack_recover_idx);   /* non-speculative top-of-stack */

/* update the branch predictor, only useful for stateful predictors */
void
bpred_update(struct bpred *pred,        /* branch predictor instance */
             SS_ADDR_TYPE baddr,        /* branch address */
             SS_ADDR_TYPE btarget,      /* resolved branch target */
             int taken,                 /* non-zero if branch was taken */
             int pred_taken,            /* non-zero if branch was pred taken */
             int correct,               /* was earlier prediction correct? */
             enum ss_opcode op,         /* opcode of instruction */
             int r31p,                  /* is this using r31? */
             struct bpred_update *dir_update_ptr);/* predictor state pointer */
```

- lookup function makes a direction/address predictions
- update function updates predictor state once direction and address are known

Cache Module (cache.[hc])

```
/* create and initialize a general cache structure */
struct cache *                          /* pointer to cache created */
cache_create(char *name,                /* name of the cache */
             int nsets,                 /* total number of sets in cache */
             int bsize,                 /* block (line) size of cache */
             int balloc,                /* allocate data space for blocks? */
             int usize,                 /* size of user data to alloc w/blks */
             int assoc,                 /* associativity of cache */
             enum cache_policy policy,  /* replacement policy w/in sets */
             /* block access function, see description w/in struct cache def */
             unsigned int (*blk_access_fn)(cmd, baddr, bsize, blk, now),
             unsigned int hit_latency);/* latency in cycles for a hit */

/* register cache stats */
void cache_reg_stats(struct cache *cp, struct stat_sdb_t *sdb);
```

- ultra-vanilla cache module
  - can implement low- and high-assoc, caches, TLBs, etc...
  - good performance for all geometries
  - assumes a single-ported, fully pipelined backside bus
- `usize` specifies per block data space, access via `udata`

Cache Module (cache.[hc])

```
/* access a cache */
unsigned int                                /* latency of access in cycles */
cache_access(struct cache *cp,              /* cache to access */
             enum mem_cmd cmd,              /* access type, Read or Write */
             SS_ADDR_TYPE addr,             /* address of access */
             void *vp,                      /* ptr to buffer for input/output */
             int nbytes,                    /* number of bytes to access */
             SS_TIME_TYPE now,              /* time of access */
             char **udata,                  /* for return of user data ptr */
             SS_ADDR_TYPE *repl_addr);      /* for address of replaced block */

/* return non-zero if block containing address ADDR is contained in cache */
int cache_probe(struct cache *cp, SS_ADDR_TYPE addr);

/* flush the entire cache */
unsigned int                                /* latency of the flush operation */
cache_flush(struct cache *cp,               /* cache instance to flush */
            SS_TIME_TYPE now);              /* time of cache flush */

/* flush the block containing ADDR from the cache CP */
unsigned int                                /* latency of flush operation */
cache_flush_addr(struct cache *cp,          /* cache instance to flush */
                 SS_ADDR_TYPE addr,         /* address of block to flush */
                 SS_TIME_TYPE now);         /* time of cache flush */
```

- set `now` to zero if no timing model

Example Cache Miss Handler

```
/* l1 data cache l1 block miss handler function */
static unsigned int                        /* latency of block access */
dl1_access_fn(enum mem_cmd cmd,            /* access cmd, Read or Write */
              SS_ADDR_TYPE baddr,          /* block address to access */
              int bsize,                   /* size of block to access */
              struct cache_blk *blk,       /* ptr to block in upper level */
              SS_TIME_TYPE now)            /* time of access */
{
  unsigned int lat;

  if (cache_dl2) { /* access next level of data cache hierarchy */
      lat = cache_access(cache_dl2, cmd, baddr, NULL, bsize, now, NULL, NULL);
      if (cmd == Read)
          return lat;
      else
          return 0;
    }
  else { /* access main memory */
      if (cmd == Read)
          return mem_access_latency(bsize);
      else
          return 0;
    }
}
```

- L1 D-cache miss handler, accesses L2 or main memory

Tutorial Overview

- Overview and Basics
- How to Use the SimpleScalar Architecture and Simulators
- How to Use SIM-OUTORDER
- How to Build Your Own Simulators
- How to Modify the ISA
- How to Use the Memory Hierarchy Extensions
- Limitations of the Tools
- Wrapup

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” directory 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, which may cause great pain!

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 Austin's dissertation work, does exactly this

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)`

Backups

To Get Plugged In

- SimpleScalar public releases available from UW-Madison
  - Public Release 2 is available from: `http://www.cs.wisc.edu/~mscalar/simplescalar.html`
- Technical Report:
  - "*Evaluating Future Microprocessors: the SimpleScalar Tools Set*", UW-Madison Tech Report #1308, July 1996
- SimpleScalar mailing list:
  - `simplescalar@cs.wisc.edu`
  - contact Doug Burger (`dburger@cs.wisc.edu`) to join

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
    - 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

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-Profile: Program Profiling Simulator

- generates program profiles, by symbol and by address
- extra options:

`-iclass`	- instruction class profiling (e.g., ALU, branch)
`-iprof`	- instruction profiling (e.g., bnez, addi, etc...)
`-brprof`	- branch class profiling (e.g., direct, calls, cond)
`-amprof`	- address mode profiling (e.g., displaced, R+R)
`-segprof`	- load/store segment profiling (e.g., data, heap)
`-tsymprof`	- execution profile by text symbol (i.e., funcs)
`-dsymprof`	- reference profile by data segment symbol
`-taddrprof`	- execution profile by text address
`-all`	- enable all of the above options
`-pcstat <stat>`	- record statistic `<stat>` by text address

- NOTE: “`-taddrprof`” `==` “`-pcstat sim_num_insn`”

Sim-Cache: Multi-level Cache Simulator

- generates one- and two-level cache hierarchy statistics and profiles
- extra options (also supported on sim-outorder):

`-cache:dl1 <config>`	- level 1 data cache configuration
`-cache:dl2 <config>`	- level 2 data cache configuration
`-cache:il1 <config>`	- level 1 instruction cache configuration
`-cache:il2 <config>`	- level 2 instruction cache configuration
`-tlb:dtlb <config>`	- data TLB configuration
`-tlb:itlb <config>`	- instruction TLB configuration
`-flush <config>`	- flush caches on system calls
`-icompress`	- remaps 64-bit inst addresses to 32-bit equiv.
`-pcstat <stat>`	- record statistic `<stat>` by text address

Specifying Cache Configurations

- all caches and TLB configurations specified with same format:

```
<name>:<nsets>:<bsize>:<assoc>:<repl>
```

- where:

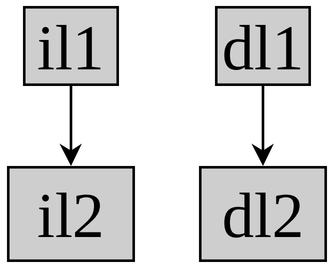
`<name>`	- cache name (make this unique)
`<nsets>`	- number of sets
`<assoc>`	- associativity (number of “ways”)
`<repl>`	- set replacement policy
	`l` - for LRU
	`f` - for FIFO
	`r` - for RANDOM

- examples:

`il1:1024:32:2:l`	2-way set-assoc 64k-byte cache, LRU
`dtlb:1:4096:64:r`	64-entry fully assoc TLB w/ 4k pages, random replacement

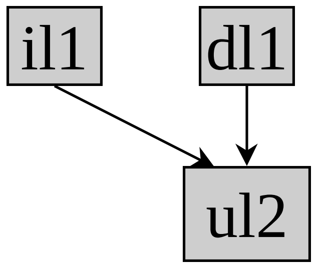
Specifying Cache Hierarchies

- specify all cache parameters in no unified levels exist, e.g.,



```
-cache:il1 il1:128:64:1:l -cache:il2 il2:128:64:4:l
-cache:dl1 dl1:256:32:1:l -cache:dl2 dl2:1024:64:2:l
```

- to unify any level of the hierarchy, "point" an I-cache level into the data cache hierarchy:



```
-cache:il1 il1:128:64:1:l -cache:il2 dl2
-cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l
```

Sim-Cheetah: Multi-Config Cache Simulator

- generates cache statistics and profiles for multiple cache configurations in a single program execution
- uses Cheetah cache simulation engine
  - ❑ written by Rabin Sugumar and Santosh Abraham while at Umich
  - ❑ modified to be a standalone library, see "`libcheetah/`" directory
- extra options:

`-refs {inst,data,unified}`	- specify reference stream to analyze
`-C {fa,sa,dm}`	- cache config. i.e., fully or set-assoc or direct
`-R {lru, opt}`	- replacement policy
`-a <sets>`	- log base 2 number of set in minimum config
`-b <sets>`	- log base 2 number of set in maximum config
`-l <line>`	- cache line size in bytes
`-n <assoc>`	- maximum associativity to analyze (log base 2)
`-in <interval>`	- cache size interval for fully-assoc analyses
`-M <size>`	- maximum cache size of interest
`-c <size>`	- cache size for direct-mapped analyses

Sim-Outorder: Detailed Performance Simulator

- generates timing statistics for a detailed out-of-order issue processor core with two-level cache memory hierarchy and main memory
- extra options:

`-fetch:ifqsize <size>`	- instruction fetch queue size (in insts)
`-fetch:mplat <cycles>`	- extra branch mis-prediction latency (cycles)
`-bpred <type>`	- specify the branch predictor
`-decode:width <insts>`	- decoder bandwidth (insts/cycle)
`-issue:width <insts>`	- RUU issue bandwidth (insts/cycle)
`-issue:inorder`	- constrain instruction issue to program order
`-issue:wrongpath`	- permit instruction issue after mis-speculation
`-ruu:size <insts>`	- capacity of RUU (insts)
`-lsq:size <insts>`	- capacity of load/store queue (insts)
`-cache:dl1 <config>`	- level 1 data cache configuration
`-cache:dl1lat <cycles>`	- level 1 data cache hit latency

Sim-Outorder: Detailed Performance Simulator

`-cache:dl2 <config>`	- level 2 data cache configuration
`-cache:dl2lat <cycles>`	- level 2 data cache hit latency
`-cache:il1 <config>`	- level 1 instruction cache configuration
`-cache:il1lat <cycles>`	- level 1 instruction cache hit latency
`-cache:il2 <config>`	- level 2 instruction cache configuration
`-cache:il2lat <cycles>`	- level 2 instruction cache hit latency
`-cache:flush`	- flush all caches on system calls
`-cache:icompress`	- remap 64-bit inst addresses to 32-bit equiv.
`-mem:lat <1st> <next>`	- specify memory access latency (first, rest)
`-mem:width`	- specify width of memory bus (in bytes)
`-tlb:itlb <config>`	- instruction TLB configuration
`-tlb:dtlb <config>`	- data TLB configuration
`-tlb:lat <cycles>`	- latency (in cycles) to service a TLB miss

Sim-Outorder: Detailed Performance Simulator

`-res:ialu`	- specify number of integer ALUs
`-res:imult`	- specify number of integer multiplier/dividers
`-res:memports`	- specify number of first-level cache ports
`-res:fpalu`	- specify number of FP ALUs
`-res:fpmult`	- specify number of FP multiplier/dividers
`-pcstat <stat>`	- record statistic `<stat>` by text address
`-ptrace <file> <range>`	- generate pipetrace

Specifying the Branch Predictor

- specifying the branch predictor type:

```
-bpred <type>
```

the supported predictor types are:

`nottaken`	always predict not taken
`taken`	always predict taken
`perfect`	perfect predictor
`bimod`	bimodal predictor (BTB w/ 2 bit counters)
`2lev`	2-level adaptive predictor

- configuring the bimodal predictor (only useful when "`-bpred bimod`" is specified):

`-bpred:bimod <size>`	size of direct-mapped BTB

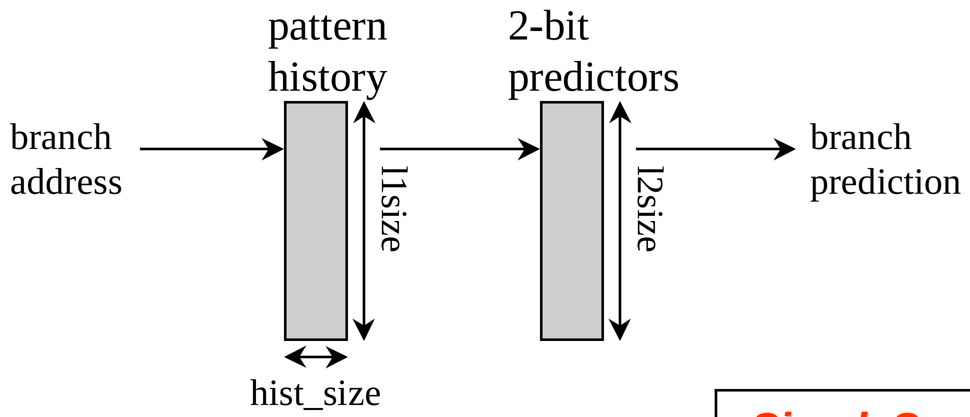
Specifying the Branch Predictor (cont.)

- configuring the 2-level adaptive predictor (only useful when "`-bpred 2lev`" is specified):

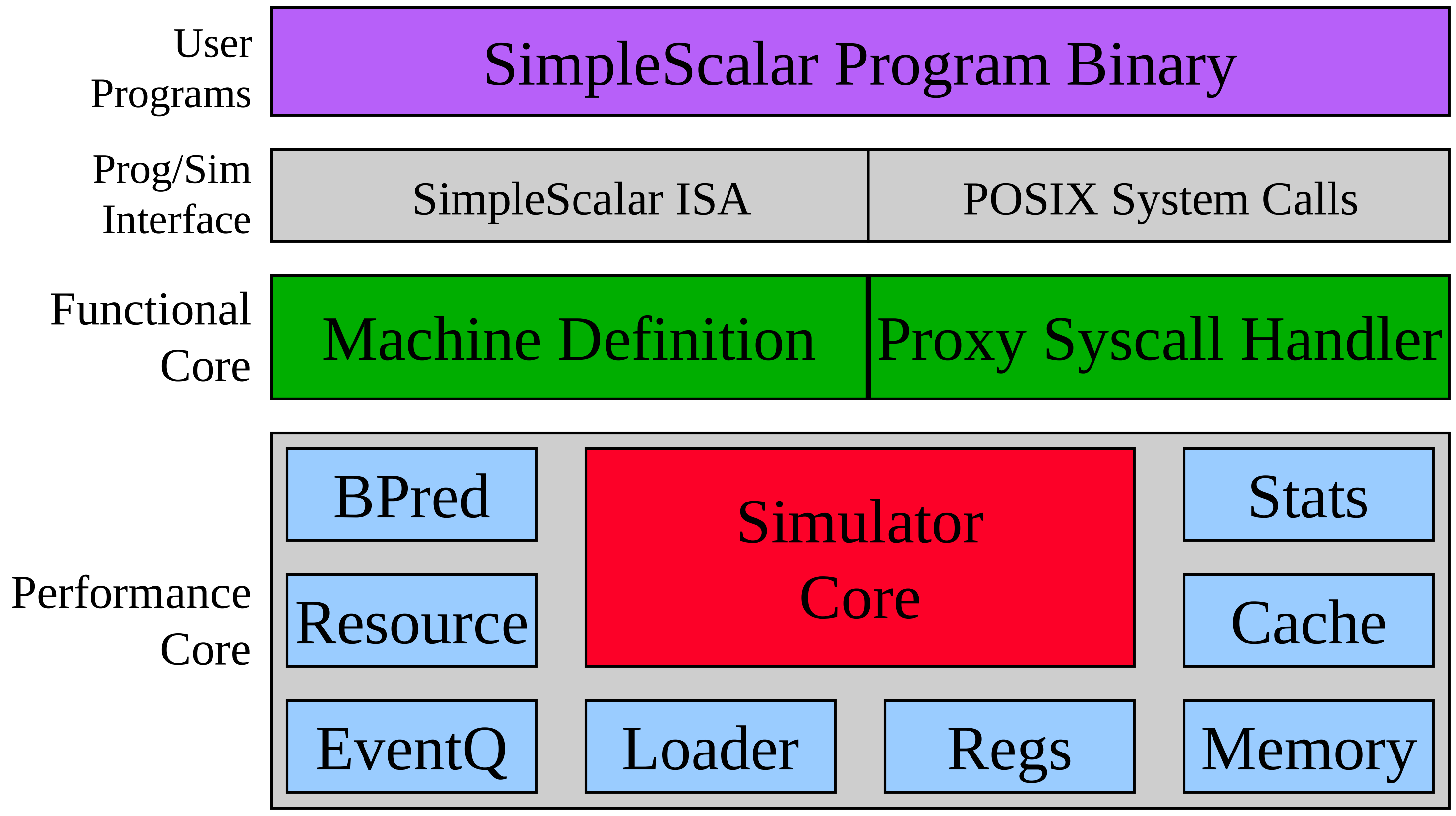
```
-bpred:2lev <l1size> <l2size> <hist_size>
```

where:

`<l1size>`	size of the first level table
`<l2size>`	size of the second level table
`<hist_size>`	history (pattern) width



Simulator Structure



- modular components facilitate "rolling your own"
- performance core is optional

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

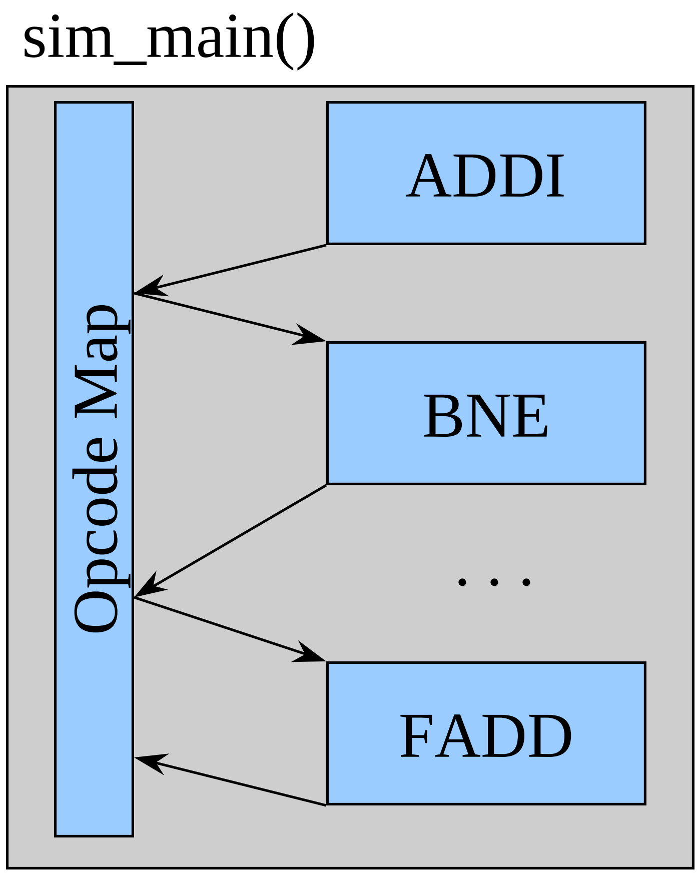
Experiences and Insights

- 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

Fast Functional Simulator


sim_main()
Opcode Map
ADDI
BNE
. . .
FADD