Flexim Simulator User Guide

(2010.9. Pre-release, Work-in-Progress)

Min Cai

Beijing Institute of Technology

Contents

Part I. Introduction

1 About Flexim

Flexim is an open-source, modular and highly configurable architectural simulator for evaluating emerging multicore processors. It can run statically compiled MIPS32 Little-Endian (LE) programs.

For the latest Flexim code, please visit the project's website on Github: http://github.com/mcai/flexim.

2 Key Features

1. Architectural
   - Simulation of a classic five-stage superscalar pipeline with out-of-order execution.
   - Multi-level memory hierarchy with the directory-based MESI cache coherence protocol.
   - Support for Syscall-emulation mode simulation (i.e., application only, no need to boot an OS).
   - Correct execution of several state-of-the-art benchmark suites, e.g., wcet_bench, Olden and CPU2006.
2. Non-architectural
   - Developed from scratch in the object-oriented system programming language D 2.0. Great efforts are made to advocate software engineering practices in the simulator construction.
   - A powerful infrastructure that provides common functionalities such as eventing, logging and XML I/O.
   - Pervasive use of XML-based I/O for architectural, workload and experiment configurations and statistics.
   - Easy to use. No scripting. Only required are a statically compiled simulator executable and a few XML files.

3 System Requirements

1. Make sure that you have a Ubuntu 10.04 Linux machine. Other popular Linux distributions may work as well if you are lucky enough.
2. Make sure that you have the latest DMD 2.0 compiler installed. If not, go to this page and download "dmd D 2.0 compiler 1-click install for Ubuntu": http://www.digitalmars.com/d/download.html.

4 How to Build and Run Flexim

1. Unpack the zip or tar file containing the Flexim source.
2. In the main directory of the distribution, you can
   - build Flexim using the command: '`make`';
   - remove all the built files using the command: '`make clean`'.

   By default, the flexim binary is placed in the bin/ folder.
3. Download and unpack cross-compiler-mipsel.tar.bz2 from http://github.com/mcai/flexim/downloads/. Use it to compile MIPS32 LE programs to be simulated by Flexim.
4. In the subdirectory build/, you can start simulation with the default simulation configuration using the command: "`./flexim`" or "`./flexim --experiment=<experiment-name>`". Benchmarks and experiments are specified in the subdirectory configs/benchmarks/ and configs/experiments/, respectively.
5. You can find configuration and statistics files in the configs/ and stats/ subdirectories, respectively. Some sample XML files are provided for your reference.
6. Useful tip: As with all other open source projects, you can learn more by digging into the Flexim source code.

5 Contact Information

If you have any questions, please feel free to contact: Min Cai <itecgo@163.com>.

Part II. Design Documentation

6 Overview

The whole development of the Flexim simulator encompasses three main categories of functionalities: functional simulation, performance simulation and supporting infrastructure.

7 Development Progress

Main Category	Current Progress	
Functional Simulation	Int. Inst. Decoding & Execution	OK for wcet-bench, mst, em3d, etc.
	Fp. Inst. Decoding & Execution	OK for wcet-bench, mst, em3d, etc.
	System Call Emulation	OK for wcet-bench, mst, em3d, etc.
	MIPS LE ELF Exe. Loader	Can run statically compiled programs
Performance Simulation	Five-stage OoO pipelining	RUU-based; to be written
	Set-associative cache structure	OK
	Cache coherence	Being rewritten; in good progress
	On-chip interconnect	Planned
	Interface to external DRAM simulators	To be planned
Supporting Infrastructure	Eventing and callback mechanisms	OK, pervasive use in existing code
	Categorized logging mechanism	OK, limited use in existing code
	XML-based I/O for configs and stats	OK
	Plotting and table generation for experiments	Planned

8 Functional Simulation

8.1 Instruction Decoding and Execution

In Flexim, there are two kinds of instructions, i.e., static instructions and dynamic instructions.

8.1.1 Basic Instructions

1. nop.

```
```

2. syscall.

```
thread.syscall(thread.intRegs[2]);
```

3. sll.

```
thread.intRegs[this[RD]] = thread.intRegs[this[RT]] << this[SA];
```

4. sllv.

```
thread.intRegs[this[RD]] = thread.intRegs[this[RT]] << bits(thread.intRegs[this[RS]], 4, 0);
```

5. sra.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RT]] >> this[SA];
```

6. srav.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RT]]
                                >> bits(thread.intRegs[this[RS]], 4, 0);
```

7. srl.

```
thread.intRegs[this[RD]] = cast(uint) thread.intRegs[this[RT]] >> this[SA];
```

8. srlv.

```
thread.intRegs[this[RD]] = cast(uint) thread.intRegs[this[RT]]
                                >> bits(thread.intRegs[this[RS]], 4, 0);
```

8.1.2 Branching Instructions

1. Common operations found in the implementation of branching operations.
   Displacement calculation:

```
this.displacement = sext(this[OFFSET] << 2, 16);
```

   Branching function:

```
thread.nnpc = thread.npc + this.displacement;
```

2. b.

```
this.branch(thread);
```

3. bal.

```
thread.intRegs[ReturnAddressReg] = thread.nnpc;
this.branch(thread);
```

4. beq.

```
if(cast(int) thread.intRegs[this[RS]] == cast(int) thread.intRegs[this[RT]]) {
        this.branch(thread);
}
```

5. beqz.

```
if(cast(int) thread.intRegs[this[RS]] == 0) {
        this.branch(thread);
}
```

6. bgez.

```
if(cast(int) thread.intRegs[this[RS]] >= 0) {
        this.branch(thread);
}
```

7. bgezal.

```
thread.intRegs[ReturnAddressReg] = thread.nnpc;
if(cast(int) thread.intRegs[this[RS]] >= 0) {
        this.branch(thread);
}
```

8. bgtz.

```
if(cast(int) thread.intRegs[this[RS]] > 0) {
        this.branch(thread);
}
```

9. blez.

```
if(cast(int) thread.intRegs[this[RS]] <= 0) {
        this.branch(thread);
}
```

10. bltz.

```
if(cast(int) thread.intRegs[this[RS]] < 0) {
        this.branch(thread);
}
```

11. bltzal.

```
thread.intRegs[ReturnAddressReg] = thread.nnpc;
if(cast(int) thread.intRegs[this[RS]] < 0) {
        this.branch(thread);
}
```

12. bne.

```
if(cast(int) thread.intRegs[this[RS]] != cast(int) thread.intRegs[this[RT]]) {
        this.branch(thread);
}
```

13. bnez.

```
if(cast(int) thread.intRegs[this[RS]] != 0) {
        this.branch(thread);
}
```

14. bc1f.

```
uint fcsr = thread.miscRegs.fcsr;
bool cond = getFCC(fcsr, this[BRANCH_CC]) == 0;

if(cond) {
        this.branch(thread);
}
```

15. bc1t.

```
uint fcsr = thread.miscRegs.fcsr;
bool cond = getFCC(fcsr, this[BRANCH_CC]) == 1;

if(cond) {
        this.branch(thread);
}
```

16. bc1fl.

```
uint fcsr = thread.miscRegs.fcsr;
bool cond = getFCC(fcsr, this[BRANCH_CC]) == 0;

if(cond) {
        this.branch(thread);
}
else {
        thread.npc = thread.nnpc;
        thread.nnpc = thread.nnpc + uint.sizeof;
}
```

17. bc1tl.

```
uint fcsr = thread.miscRegs.fcsr;
bool cond = getFCC(fcsr, this[BRANCH_CC]) == 1;

if(cond) {
        this.branch(thread);
}
else {
        thread.npc = thread.nnpc;
        thread.nnpc = thread.nnpc + uint.sizeof;
}
```

8.1.3 Jumping Instructions

1. Common operations found in the implementation of jumping operations.
   Abstract definition of target PC calculation:

```
abstract uint targetPc(Thread thread);
```

   Jumping function:

```
thread.nnpc = addr;
```

2. j.
   Target PC calculation:

```
return mbits(thread.npc, 32, 28) | this.target;
```

   Execution:

```
this.jump(thread, this.targetPc(thread));
```

3. jal.
   Target PC calculation:

```
return mbits(thread.npc, 32, 28) | this.target;
```

   Execution:

```
thread.intRegs[ReturnAddressReg] = thread.nnpc;
this.jump(thread, this.targetPc(thread));
```

4. jalr.
   Target PC calculation:

```
return thread.intRegs[this[RS]];
```

   Execution:

```
thread.intRegs[this[RD]] = thread.nnpc;
this.jump(thread, this.targetPc(thread));
```

5. jr.
   Target PC calculation:

```
return thread.intRegs[this[RS]];
```

   Execution:

```
this.jump(thread, this.targetPc(thread));
```

8.1.4 Floating Point Arithmetic Instructions

1. add_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
double ft = thread.floatRegs.getDouble(this[FT]);

double fd = fs + ft;

thread.floatRegs.setDouble(fd, this[FD]);
```

2. sub_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
double ft = thread.floatRegs.getDouble(this[FT]);

double fd = fs - ft;

thread.floatRegs.setDouble(fd, this[FD]);
```

3. mul_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
double ft = thread.floatRegs.getDouble(this[FT]);

double fd = fs * ft;

thread.floatRegs.setDouble(fd, this[FD]);
```

4. div_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
double ft = thread.floatRegs.getDouble(this[FT]);

double fd = fs / ft;

thread.floatRegs.setDouble(fd, this[FD]);
```

5. sqrt_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);

double fd = sqrt(fs);

thread.floatRegs.setDouble(fd, this[FD]);
```

6. abs_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);

double fd = fabs(fs);

thread.floatRegs.setDouble(fd, this[FD]);
```

7. neg_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);

double fd = -1 * fs;

thread.floatRegs.setDouble(fd, this[FD]);
```

8. mov_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
double fd = fs;

thread.floatRegs.setDouble(fd, this[FD]);
```

9. add_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
float ft = thread.floatRegs.getFloat(this[FT]);

float fd = fs + ft;

thread.floatRegs.setFloat(fd, this[FD]);
```

10. sub_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
float ft = thread.floatRegs.getFloat(this[FT]);

float fd = fs - ft;

thread.floatRegs.setFloat(fd, this[FD]);
```

11. mul_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
float ft = thread.floatRegs.getFloat(this[FT]);

float fd = fs * ft;

thread.floatRegs.setFloat(fd, this[FD]);
```

12. div_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
float ft = thread.floatRegs.getFloat(this[FT]);

float fd = fs / ft;

thread.floatRegs.setFloat(fd, this[FD]);
```

13. sqrt_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);

float fd = sqrt(fs);

thread.floatRegs.setFloat(fd, this[FD]);
```

14. abs_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);

float fd = fabs(fs);

thread.floatRegs.setFloat(fd, this[FD]);
```

15. neg_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);

float fd = -fs;

thread.floatRegs.setFloat(fd, this[FD]);
```

16. mov_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
float fd = fs;

thread.floatRegs.setFloat(fd, this[FD]);
```

17. cvt_d_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
double fd = cast(double) fs;

thread.floatRegs.setDouble(fd, this[FD]);
```

18. cvt_w_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
uint fd = cast(uint) fs;

thread.floatRegs.setUint(fd, this[FD]);
```

19. cvt_l_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
ulong fd = cast(ulong) fs;

thread.floatRegs.setUlong(fd, this[FD]);
```

20. cvt_s_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
float fd = cast(float) fs;

thread.floatRegs.setFloat(fd, this[FD]);
```

21. cvt_w_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
uint fd = cast(uint) fs;

thread.floatRegs.setUint(fd, this[FD]);
```

22. cvt_l_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
ulong fd = cast(ulong) fs;

thread.floatRegs.setUlong(fd, this[FD]);
```

23. cvt_s_w.

```
uint fs = thread.floatRegs.getUint(this[FS]);
float fd = cast(float) fs;

thread.floatRegs.setFloat(fd, this[FD]);
```

24. cvt_d_w.

```
uint fs = thread.floatRegs.getUint(this[FS]);
double fd = cast(double) fs;

thread.floatRegs.setDouble(fd, this[FD]);
```

25. cvt_s_l.

```
ulong fs = thread.floatRegs.getUlong(this[FS]);
float fd = cast(float) fs;

thread.floatRegs.setFloat(fd, this[FD]);
```

26. cvt_d_l.

```
ulong fs = thread.floatRegs.getUlong(this[FS]);
double fd = cast(double) fs;

thread.floatRegs.setDouble(fd, this[FD]);
```

27. c_<cond>_d type instructions, which include c_f_d, c_un_d, c_eq_d, c_ueq_d, c_olt_d, c_ult_d, c_ole_d, c_ule_d, c_sf_d, c_ngle_d, c_seq_d, c_ngl_d, c_lt_d, c_nge_d, c_le_d and c_ngt_d.

```
double fs = thread.floatRegs.getDouble(this[FS]);
double ft = thread.floatRegs.getDouble(this[FT]);
uint fcsr = thread.miscRegs.fcsr;

bool less;
bool equal;

bool unordered = isnan(fs) || isnan(ft);
if(unordered) {
        equal = false;
        less = false;
}
else {
        equal = fs == ft;
        less = fs < ft;
}

uint cond = this[COND];

if(((cond&0x4) && less)||((cond&0x2) && equal)||((cond&0x1) && unordered)) {
        setFCC(fcsr, this[CC]);
}
else {
        clearFCC(fcsr, this[CC]);
}

thread.miscRegs.fcsr = fcsr;
```

28. c_<cond>_s type instructions, which include c_f_s, c_un_s, c_eq_s, c_ueq_s, c_olt_s, c_ult_s, c_ole_s, c_ule_s, c_sf_s, c_ngle_s, c_seq_s, c_ngl_s, c_lt_s, c_nge_s, c_le_s and c_ngt_s.

```
float fs = thread.floatRegs.getFloat(this[FS]);
float ft = thread.floatRegs.getFloat(this[FT]);
uint fcsr = thread.miscRegs.fcsr;

bool less;
bool equal;

bool unordered = isnan(fs) || isnan(ft);
if(unordered) {
        equal = false;
        less = false;
}
else {
        equal = fs == ft;
        less = fs < ft;
}

uint cond = this[COND];

if(((cond&0x4) && less)||((cond&0x2) && equal)||((cond&0x1) && unordered)) {
        setFCC(fcsr, this[CC]);
}
```

```
else {
        clearFCC(fcsr, this[CC]);
}

thread.miscRegs.fcsr = fcsr;
```

29. mfc1.

```
uint fs = thread.floatRegs.getUint(this[FS]);
thread.intRegs[this[RT]] = fs;
```

30. cfc1.

```
uint fcsr = thread.miscRegs.fcsr;

uint rt = 0;

if(this[FS] == 31) {
        rt = fcsr;
        thread.intRegs[this[RT]] = rt;
}
```

31. mtc1.

```
uint rt = thread.intRegs[this[RT]];
thread.floatRegs.setUint(rt, this[FS]);
```

32. ctc1.

```
uint rt = thread.intRegs[this[RT]];

if(this[FS]) {
        thread.miscRegs.fcsr = rt;
}
```

8.1.5 Integer Arithmetic Instructions

1. Common operations found in the implementation of integer arithmetic operations.
   imm.

```
this.imm = cast(short) machInst[INTIMM];
```

zextImm.

```
this.zextImm = 0x0000FFFF & machInst[INTIMM];
```

sextImm.

```
this.sextImm = sext(machInst[INTIMM], 16);
```

2. add.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RS]]
                                + cast(int) thread.intRegs[this[RT]];
logging.warn(LogCategory.INSTRUCTION, "Add: overflow trap not implemented.");
```

3. addi.

```
thread.intRegs[this[RT]] = cast(int) thread.intRegs[this[RS]] + this.sextImm;
logging.warn(LogCategory.INSTRUCTION, "Addi: overflow trap not implemented.");
```

4. addiu.

```
thread.intRegs[this[RT]] = cast(int) thread.intRegs[this[RS]] + this.sextImm;
```

5. addu.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RS]]
                         + cast(int) thread.intRegs[this[RT]];
```

6. sub.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RS]]
                         - cast(int) thread.intRegs[this[RT]];
logging.warn(LogCategory.INSTRUCTION, "Sub: overflow trap not implemented.");
```

7. subu.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RS]]
                         - cast(int) thread.intRegs[this[RT]];
```

8. and.

```
thread.intRegs[this[RD]] = thread.intRegs[this[RS]] & thread.intRegs[this[RT]];
```

9. andi.

```
thread.intRegs[this[RT]] = thread.intRegs[this[RS]] & this.zextImm;
```

10. nor.

```
thread.intRegs[this[RD]] = ~(thread.intRegs[this[RS]] | thread.intRegs[this[RT]]);
```

11. or.

```
thread.intRegs[this[RD]] = thread.intRegs[this[RS]] | thread.intRegs[this[RT]];
```

12. ori.

```
thread.intRegs[this[RT]] = thread.intRegs[this[RS]] | this.zextImm;
```

13. xor.

```
thread.intRegs[this[RD]] = thread.intRegs[this[RS]] ^ thread.intRegs[this[RT]];
```

14. xori.

```
thread.intRegs[this[RT]] = thread.intRegs[this[RS]] ^ this.zextImm;
```

15. slt.

```
thread.intRegs[this[RD]] = cast(int) thread.intRegs[this[RS]]
                         < cast(int) thread.intRegs[this[RT]] ? 1 : 0;
```

16. slti.

```
thread.intRegs[this[RT]] = cast(int) thread.intRegs[this[RS]] < this.sextImm ? 1 : 0;
```

17. sltiu.

```
thread.intRegs[this[RT]] = cast(uint) thread.intRegs[this[RS]] < this.zextImm ? 1 : 0;
```

18. sltu.

```
thread.intRegs[this[RD]] = cast(uint) thread.intRegs[this[RS]]
                         < cast(uint) thread.intRegs[this[RT]] ? 1 : 0;
```

19. lui.

```
thread.intRegs[this[RT]] = this.imm << 16;
```

20. divu.

```
ulong rs = 0;
ulong rt = 0;

uint lo = 0;
uint hi = 0;

rs = thread.intRegs[this[RS]];
rt = thread.intRegs[this[RT]];

if(rt != 0) {
        lo = cast(uint) (rs / rt);
        hi = cast(uint) (rs % rt);
}

thread.miscRegs.lo = lo;
thread.miscRegs.hi = hi;
```

21. div.

```
long rs = 0;
long rt = 0;

uint lo = 0;
uint hi = 0;

rs = sext(thread.intRegs[this[RS]], 32);
rt = sext(thread.intRegs[this[RT]], 32);

if(rt != 0) {
        lo = cast(uint) (rs / rt);
        hi = cast(uint) (rs % rt);
}

thread.miscRegs.lo = lo;
thread.miscRegs.hi = hi;
```

22. mflo.

```
thread.intRegs[this[RD]] = thread.miscRegs.lo;
```

23. mfhi.

```
thread.intRegs[this[RD]] = thread.miscRegs.hi;
```

24. mtlo.

```
thread.miscRegs.lo = thread.intRegs[this[RD]];
```

25. mthi.

```
thread.miscRegs.hi = thread.intRegs[this[RD]];
```

26. mult.

```
long rs = 0;
long rt = 0;

rs = sext(thread.intRegs[this[RS]], 32);
rt = sext(thread.intRegs[this[RT]], 32);
```

```
long val = rs * rt;

uint lo = cast(uint) bits64(val, 31, 0);
uint hi = cast(uint) bits64(val, 63, 32);

thread.miscRegs.lo = lo;
thread.miscRegs.hi = hi;
```

27. multu.

```
ulong rs = 0;
ulong rt = 0;

rs = thread.intRegs[this[RS]];
rt = thread.intRegs[this[RT]];

ulong val = rs * rt;

uint lo = cast(uint) bits64(val, 31, 0);
uint hi = cast(uint) bits64(val, 63, 32);

thread.miscRegs.lo = lo;
thread.miscRegs.hi = hi;
```

8.1.6 Memory Access Instructions

1. Common operations found in the implementation of memory access operations.
   displacement calculation.

```
this.displacement = sext(machInst[OFFSET], 16);
```

   basic effective address calculation.

```
return thread.intRegs[this[RS]] + this.displacement;
```

2. lb.

```
byte mem = 0;
thread.mem.readByte(this.ea(thread), cast(ubyte*) &mem);
thread.intRegs[this[RT]] = mem;
```

3. lbu.

```
ubyte mem = 0;
thread.mem.readByte(this.ea(thread), &mem);
thread.intRegs[this[RT]] = mem;
```

4. lh.

```
short mem = 0;
thread.mem.readHalfWord(this.ea(thread), cast(ushort*) &mem);
thread.intRegs[this[RT]] = mem;
```

5. lhu.

```
ushort mem = 0;
thread.mem.readHalfWord(this.ea(thread), &mem);
thread.intRegs[this[RT]] = mem;
```

6. lw.

```
int mem = 0;
thread.mem.readWord(this.ea(thread), cast(uint*) &mem);
thread.intRegs[this[RT]] = mem;
```

7. lwl.
   overriden effective address calculation.

```
uint addr = thread.intRegs[this[RS]] + this.displacement;
return addr & ~3;
```

```
uint addr = thread.intRegs[this[RS]] + this.displacement;

uint ea = addr & ~3;
uint byte_offset = addr & 3;

uint mem = 0;

thread.mem.readWord(ea, &mem);

uint mem_shift = 24 - 8 * byte_offset;

uint rt = (mem << mem_shift) | (thread.intRegs[this[RT]] & mask(mem_shift));

thread.intRegs[this[RT]] = rt;
```

8. lwr.
   overriden effective address calculation.

```
uint addr = thread.intRegs[this[RS]] + this.displacement;
return addr & ~3;
```

```
uint addr = thread.intRegs[this[RS]] + this.displacement;

uint ea = addr & ~3;
uint byte_offset = addr & 3;

uint mem = 0;

thread.mem.readWord(ea, &mem);

uint mem_shift = 8 * byte_offset;

uint rt = (thread.intRegs[this[RT]] & (mask(mem_shift) << (32 - mem_shift)))
                                | (mem >> mem_shift);

thread.intRegs[this[RT]] = rt;
```

9. ll.

```
uint mem = 0;
thread.mem.readWord(this.ea(thread), &mem);
thread.intRegs[this[RT]] = mem;
```

10. lwc1.

```
uint mem = 0;
thread.mem.readWord(this.ea(thread), &mem);
thread.floatRegs.setUint(mem, this[FT]);
```

11. ldc1.

```
ulong mem = 0;
thread.mem.readDoubleWord(this.ea(thread), &mem);
thread.floatRegs.setUlong(mem, this[FT]);
```

12. sb.

```
ubyte mem = cast(ubyte) bits(thread.intRegs[this[RT]], 7, 0);
thread.mem.writeByte(this.ea(thread), mem);
```

13. sh.

```
ushort mem = cast(ushort) bits(thread.intRegs[this[RT]], 15, 0);
thread.mem.writeHalfWord(this.ea(thread), mem);
```

14. sw.

```
uint mem = thread.intRegs[this[RT]];
thread.mem.writeWord(this.ea(thread), mem);
```

15. swl.
overriden effective address calculation.

```
uint addr = thread.intRegs[this[RS]] + this.displacement;
return addr & ~3;
```

```
uint addr = thread.intRegs[this[RS]] + this.displacement;

uint ea = addr & ~3;
uint byte_offset = addr & 3;

uint mem = 0;

thread.mem.readWord(ea, &mem);

uint reg_shift = 24 - 8 * byte_offset;
uint mem_shift = 32 - reg_shift;

mem = (mem & (mask(reg_shift) << mem_shift)) | (thread.intRegs[this[RT]] >> reg_shift);

thread.mem.writeWord(ea, mem);
```

16. swr.
overriden effective address calculation.

```
uint addr = thread.intRegs[this[RS]] + this.displacement;
return addr & ~3;
```

```
uint addr = thread.intRegs[this[RS]] + this.displacement;

uint ea = addr & ~3;
uint byte_offset = addr & 3;

uint mem = 0;

thread.mem.readWord(ea, &mem);

uint reg_shift = 8 * byte_offset;

mem = thread.intRegs[this[RT]] << reg_shift | (mem & (mask(reg_shift)));

thread.mem.writeWord(ea, mem);
```

17. sc.

```
uint rt = thread.intRegs[this[RT]];
thread.mem.writeWord(this.ea(thread), rt);
thread.intRegs[this[RT]] = 1;
```

18. swc1.

```
uint ft = thread.floatRegs.getUint(this[FT]);
thread.mem.writeWord(this.ea(thread), ft);
```

19. sdc1.

```
ulong ft = thread.floatRegs.getUlong(this[FT]);
thread.mem.writeDoubleWord(this.ea(thread), ft);
```

8.2 System Call Emulation

A few system calls are emulated for the correct execution of the whole wcet_bench benchmark suite, and mst and em3d from the Olden benchmark suite.

1. exit.

```
logging.haltf(LogCategory.SYSCALL, "target called exit(%d)", thread.getSyscallArg(0) & 0xff);
return 1;
```

2. read.

```
int fd = thread.getSyscallArg(0);
uint buf_addr = thread.getSyscallArg(1);
size_t count = thread.getSyscallArg(2);

void* buf = malloc(count);
ssize_t ret = core.sys.posix.unistd.read(fd, buf, count);
if(ret > 0) {
        thread.mem.writeBlock(buf_addr, ret, cast(ubyte*) buf);
}
free(buf);

return ret;
```

3. write.

```
int fd = thread.getSyscallArg(0);
uint buf_addr = thread.getSyscallArg(1);
size_t count = thread.getSyscallArg(2);

void* buf = malloc(count);
thread.mem.readBlock(buf_addr, count, cast(ubyte*) buf);
ssize_t ret = core.sys.posix.unistd.write(fd, buf, count);
free(buf);

return ret;
```

4. open.

```
char path[MAXBUFSIZE];

uint addr = thread.getSyscallArg(0);
uint tgtFlags = thread.getSyscallArg(1);
uint mode = thread.getSyscallArg(2);

int strlen = thread.mem.readString(addr, MAXBUFSIZE, &path[0]);

// translate open flags
int hostFlags = 0;
foreach(t; openFlagTable) {
        if(tgtFlags & t.tgtFlag) {
                tgtFlags &= ~t.tgtFlag;
                hostFlags |= t.hostFlag;
        }
}

// any target flags left?
if(tgtFlags != 0)
                logging.fatalf(LogCategory.SYSCALL,
                        "Syscall: open: cannot decode flags 0x%x", tgtFlags);

// Adjust path for current working directory
path = thread.process.fullPath(to!(string)(path));

// open the file
int fd = open(path.ptr, hostFlags, mode);
return fd;
```

5. close.

```
int fd = thread.getSyscallArg(0);
int ret = close(fd);
return ret;
```

6. lseek.

```
int fildes = thread.getSyscallArg(0);
off_t offset = thread.getSyscallArg(1);
int whence = thread.getSyscallArg(2);

off_t ret = lseek(fildes, offset, whence);
return ret;
```

7. getpid.

```
return thread.process.pid;
```

8. getuid.

```
return thread.process.uid;
```

9. brk.

```
uint oldbrk, newbrk;
uint oldbrk_rnd, newbrk_rnd;

newbrk = thread.getSyscallArg(0);
oldbrk = thread.process.brk;

if(newbrk == 0) {
        return thread.process.brk;
}

newbrk_rnd = Rounding!(uint).roundUp(newbrk, MEM_PAGESIZE);
oldbrk_rnd = Rounding!(uint).roundUp(oldbrk, MEM_PAGESIZE);

if(newbrk > oldbrk) {
        thread.mem.map(oldbrk_rnd, newbrk_rnd - oldbrk_rnd,
                MemoryAccessType.READ | MemoryAccessType.WRITE);
} else if(newbrk < oldbrk) {
        thread.mem.unmap(newbrk_rnd, oldbrk_rnd - newbrk_rnd);
}
thread.process.brk = newbrk;

return thread.process.brk;
```

10. getgid.

```
return thread.process.gid;
```

11. geteuid.

```
return thread.process.euid;
```

12. getegid.

```
return thread.process.egid;
```

13. fstat.

```
int fd = thread.getSyscallArg(0);
uint buf_addr = thread.getSyscallArg(1);
stat_t* buf = cast(stat_t*)(malloc(stat_t.sizeof));
int ret = fstat(fd, buf);
if(ret >= 0) {
        thread.mem.writeBlock(buf_addr, stat_t.sizeof, cast(ubyte*) buf);
}
free(buf);
return ret;
```

14. uname.

```
utsname un = {"Linux", "sim", "2.6", "Tue␣Apr␣5␣12:21:57␣UTC␣2005", "mips"};
thread.mem.writeBlock(thread.getSyscallArg(0), un.sizeof, cast(ubyte*) &un);
return 0;
```

15. _llseek.

```
int fd = thread.getSyscallArg(0);
uint offset_high = thread.getSyscallArg(1);
uint offset_low = thread.getSyscallArg(2);
uint result_addr = thread.getSyscallArg(3);
int whence = thread.getSyscallArg(4);

int ret;

if(offset_high == 0) {
        off_t lseek_ret = lseek(fd, offset_low, whence);
        if(lseek_ret >= 0) {
                ret = 0;
        }
        else {
                ret = -1;
        }
}
else {
        ret = -1;
}

return ret;
```

8.3 MIPS Little-Endian ELF Executable Loader

9 Performance Simulation

9.1 Five-Stage Out-of-Order Pipelining

A classic five-stage out-of-order issue processor core is modeled after the SimpleScalar implementation. Methods in class OoOThread implementing the pipeline stages are outlined below.

Method Name	Insts Transfer between Queues	Comments
commit()	RUU $\longrightarrow$ <committed>	Retiring insts, EAs $\dashrightarrow$ LSQ
writeback()	EventQ $\dashrightarrow$ ReadyQ	Resolving reg deps
refreshLsq()	LSQ $\longrightarrow$ ReadyQ	Resolving mem deps
issue()	ReadyQ $\longrightarrow$ EventQ	Accessing FUs and data caches
dispatch()	FetchQ $\longrightarrow$ RUU + LSQ + ReadyQ	Resolving reg deps
fetch()	ICache $\longrightarrow$ FetchQ	Fetching and decoding insts

9.2 Set-Associative Cache Structure

9.3 Cache Coherence

9.4 On-Chip Interconnect

9.5 Interface to External DRAM Simulators

10 Supporting Infrastructure

There are various supporting modules aside the aforementioned main components to advocate the reusability of the simulator, in which the ELF program loader component is used to load statically compiled MIPS32 little-endian executable into the simulator, the event queue component is used extensively to event driven the simulator per cycle, and the logging component supports configurable logging functionalities that can facilitate development and even be useful after release.

10.1 Eventing and Callback Mechanisms

10.2 Categorized Logging Mechanism

10.3 XML-Based Input/Output for Configurations and Statistics

10.4 Plotting and Table Generation for Experiments

Part III. Evaluation and Comparisons to Other Simulators

11 Benchmark Evaluation

11.1 Criteria

11.2 Results

12 Comparison to Other Simulators

12.1 Results