The Flexim User Guide

(Pre-release, Documentation-in-Progress)

Min Cai <itecgo@163.com>

Beijing Institute of Technology
Beijing, China

September 9, 2010

Contents

Chapter 1

Introduction

Welcome to 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.

1.1 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.

1.2 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	Processor core	Being rewritten; almost complete
	Set-associative cache structure	OK
	Cache coherence	Being rewritten; almost complete
	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

1.3 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.

1.4 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.

1.5 The Structure of the File

The whole development of the Flexim simulator encompasses three main categories of functionalities: functional simulation, performance simulation and the supporting infrastructure. Chapter 2 focuses on functional simulation. Chapter 3 elaborates on performance simulation. Chapter 4 focuses on the supporting infrastructure. And Chapter 5 provides the evaluation, limitations and future work of Flexim.

Chapter 2

Functional Simulation

Functional simulation encompasses the abilities to load and parse MIPS binaries, decode and execute instructions, and emulate system calls.

2.1 ELF-Formatted MIPS Little-Endian Executable Loader

The Executable and Linkable Format (ELF) is a standard binary file format for Unix and Unix-like (such as Linux) systems. Each ELF file is made up of one ELF header, followed by file data. The file data can include:

- Program header table, describing zero or more segments
- Section header table, describing zero or more sections
- Data referred to by entries in the program header table or section header table

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Each byte in the entire file is taken by no more than one section at a time, but there can be orphan bytes, which are not covered by a section. In the normal case of a Unix executable one or more sections are enclosed in one segment.

In Flexim, the tasks of loading and parsing of ELF files are done through the classes `ELFReader` and `ELF32Binary` and a few supporting code elements. Each process in Flexim is associated with one `ELF32Binary` object.

2.2 Instruction Decoding and Execution

In Flexim, there are two kinds of instructions, i.e., static instructions and dynamic instructions. A static instruction represents a decoded instruction that fetched from memory, and a dynamic instruction represents a dynamically-scheduled instruction.

Below are the details of the implemented instructions.

2.2.1 Basic Instructions

1. nop. It does nothing.
2. syscall. Its execution logic is shown as below:

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

3. sll. Its execution logic is shown as below:

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

4. sllv. Its execution logic is shown as below:

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

5. sra. Its execution logic is shown as below:

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

6. srav. Its execution logic is shown as below:

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

7. srl. Its execution logic is shown as below:

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

8. srlv. Its execution logic is shown as below:

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

2.2.2 Branching Instructions

1. Common operations found in the implementation of branching operations.
   The displacement calculation is shown as below:

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

   And the branching function is shown as below:

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

2. b. Its execution logic is shown as below:

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

3. bal. Its execution logic is shown as below:

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

4. beq. Its execution logic is shown as below:

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

5. beqz. Its execution logic is shown as below:

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

6. bgez. Its execution logic is shown as below:

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

7. bgezal. Its execution logic is shown as below:

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

8. bgtz. Its execution logic is shown as below:

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

9. blez. Its execution logic is shown as below:

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

10. bltz. Its execution logic is shown as below:

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

11. bltzal. Its execution logic is shown as below:

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

12. bne. Its execution logic is shown as below:

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

13. bnez. Its execution logic is shown as below:

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

14. bc1f. Its execution logic is shown as below:

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

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

15. bc1t. Its execution logic is shown as below:

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

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

16. bc1fl. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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;
}
```

2.2.3 Jumping Instructions

1. Common operations found in the implementation of jumping operations.
   The abstract definition of target PC calculation is shown as below:

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

   And the jumping function is shown as below:

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

2. j.
   Its target PC calculation is shown as below:

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

   Its execution logic is shown as below:

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

3. jal.
   Its target PC calculation is shown as below:

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

   Its execution logic is shown as below:

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

4. jalr.
   Its target PC calculation is shown as below:

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

   Its execution logic is shown as below:

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

5. jr.
   Its target PC calculation is shown as below:

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

   Its execution logic is shown as below:

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

2.2.4 Floating Point Arithmetic Instructions

1. add_d. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

double fd = sqrt(fs);

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

6. abs_d. Its execution logic is shown as below:

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

double fd = fabs(fs);

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

7. neg_d. Its execution logic is shown as below:

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

double fd = -1 * fs;

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

8. mov_d. Its execution logic is shown as below:

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

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

9. add_s. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

float fd = sqrt(fs);

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

14. abs_s. Its execution logic is shown as below:

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

float fd = fabs(fs);

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

15. neg_s. Its execution logic is shown as below:

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

float fd = -fs;

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

16. mov_s. Its execution logic is shown as below:

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

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

17. cvt_d_s. Its execution logic is shown as below:

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

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

18. cvt_w_s. Its execution logic is shown as below:

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

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

19. cvt_l_s. Its execution logic is shown as below:

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

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

20. cvt_s_d. Its execution logic is shown as below:

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

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

21. cvt_w_d. Its execution logic is shown as below:

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

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

22. cvt_l_d. Its execution logic is shown as below:

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

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

23. cvt_s_w. Its execution logic is shown as below:

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

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

24. cvt_d_w. Its execution logic is shown as below:

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

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

25. cvt_s_l. Its execution logic is shown as below:

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

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

26. cvt_d_l. Its execution logic is shown as below:

```
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. The execution logic of this type of instructions is shown as below:

```
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. The execution logic of this type of instructions is shown as below:

```
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. Its execution logic is shown as below:

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

30. cfc1. Its execution logic is shown as below:

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

uint rt = 0;

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

31. mtc1. Its execution logic is shown as below:

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

32. ctc1. Its execution logic is shown as below:

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

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

2.2.5 Integer Arithmetic Instructions

1. Common operations found in the implementation of integer arithmetic operations.
   Its immmediate value is calculated as:

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

   Its zero-extended immediate value is calculated as:

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

   Its sign-extended immediate value is calculated as:

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

2. add. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

4. addiu. Its execution logic is shown as below:

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

5. addu. Its execution logic is shown as below:

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

6. sub. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

8. and. Its execution logic is shown as below:

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

9. andi. Its execution logic is shown as below:

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

10. nor. Its execution logic is shown as below:

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

11. or. Its execution logic is shown as below:

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

12. ori. Its execution logic is shown as below:

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

13. xor. Its execution logic is shown as below:

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

14. xori. Its execution logic is shown as below:

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

15. slt. Its execution logic is shown as below:

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

16. slti. Its execution logic is shown as below:

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

17. sltiu. Its execution logic is shown as below:

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

18. sltu. Its execution logic is shown as below:

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

19. lui. Its execution logic is shown as below:

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

20. divu. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

23. mfhi. Its execution logic is shown as below:

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

24. mtlo. Its execution logic is shown as below:

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

25. mthi. Its execution logic is shown as below:

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

26. mult. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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;
```

2.2.6 Memory Access Instructions

1. Common operations found in the implementation of memory access operations.
   Its displacement value is calculated as:

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

   And its effective address is calculated as (overridable):

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

2. lb. Its execution logic is shown as below:

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

3. lbu. Its execution logic is shown as below:

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

4. lh. Its execution logic is shown as below:

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

5. lhu. Its execution logic is shown as below:

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

6. lw. Its execution logic is shown as below:

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

7. lwl.
   Its overriden effective address calculation is shown as below:

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

   Its execution logic is shown as below:

```
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.
   Its overriden effective address calculation is shown as below:

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

   Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

10. lwc1. Its execution logic is shown as below:

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

11. ldc1. Its execution logic is shown as below:

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

12. sb. Its execution logic is shown as below:

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

13. sh. Its execution logic is shown as below:

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

14. sw. Its execution logic is shown as below:

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

15. swl.
    Its overriden effective address calculation is shown as below:

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

    Its execution logic is shown as below:

```
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.
    Its overriden effective address calculation is shown as below:

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

    Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

18. swc1. Its execution logic is shown as below:

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

19. sdc1. Its execution logic is shown as below:

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

2.3 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. Its execution logic is shown as below:

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

2. read. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

6. lseek. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

8. getuid. Its execution logic is shown as below:

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

9. brk. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

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

11. geteuid. Its execution logic is shown as below:

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

12. getegid. Its execution logic is shown as below:

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

13. fstat. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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. Its execution logic is shown as below:

```
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;
```

Chapter 3

Performance Simulation

Performance simulation encompasses the detailed simulation of out-of-order processor cores and multi-level memory hierarchies.

3.1 Processor

Flexim supports the mix of SMT and CMP simulation. Some processor structures are shared among threads within a processor core, and others are private to each thread.

3.1.1 Cycle-Accurate Modeling of Pipeline Structures

Flexim explicitly models the Reorder Buffer (`ReorderBuffer`), the Issue Queue (`IssueQueue`), the Load/Store Queue (`LoadStoreQueue`), separate integer, floating-point and miscellaneous register files, register renaming, and the associated rename table.

The reorder buffer is modeled as a FIFO buffer. Its entries are pushed during instruction `dispatch()` and are popped during instruction `commit()`.

The issue queue is modeled as an array of issue queue entries. Each issue queue entry (`IssueQueueEntry`) can be in one of two states: either free(`IssueQueueEntryState.FREE`) or already allocated(`IssueQueueEntry.ALLOC`). Its entries are allocated at instruction `dispatch()` and are released at `issue()` and on branch mispredictions on `recoverReorderBuffer()`.

The integer, floating point and miscellaneous physical register files are modeled separately as `intRegFile`, `fpRegFile` and `miscRegFile`, respectively. All of them contain an array of physical registers(`PhysicalRegister`). Each of the physical registers can be in one of four states:

1. the register is free (`PhysicalRegisterState.FREE`),
2. the register has been allocated to an instruction but has not yet been written to (`PhysicalRegisterState.ALLOC`),
3. the register is allocated to an instruction and the value has been written (`PhysicalRegisterState.WB`),
4. the register is in the architectural state (`PhysicalRegisterState.ARCH`).

Physical registers are allocated at the `dispatch()` stage, and deallocated at `commit()` and during branch mispredictions in `recoverReorderBuffer()`.

The integer, floating point and miscellaneous rename tables are modeled separately. They maintain the current mappings of each architectural register to a physical register.

3.1.2 Details of the Processor model and Implementation

Threads within a core maintain separate program counters, but share the fetch unit and icache. Threads within a core share the available bandwidth in the front end including fetch, decode and rename. The time slice based fetch policy is implemented for the moment, more advanced policies such as icount is left for future work. Separate branch predictors are implemented per thread.

Each thread maintains its own rename table because it has its own set of architectural registers. After renaming, instructions from all threads are dispatched into the shared issue queue.

In the issue queue, instructions from all the threads participate in the instruction `wakeup()` process and compete for the issue bandwidth in `selection()`. Instructions that are selected for issue continue to perform register file accesses. All the physical register files are shared among the threads. After register file accesses are done, instructions begin execution on the functional units, which are also shared among the threads.

Loads and stores access the shared dcache among the threads within a core. In order to maintain the correct orderings of memory accesses, the load/store queue (`LoadStoreQueue`) is used. Separate load/store queue is maintained per thread, so that an unresolved address from one thread does not prevent loads in other threads from issuing.

After execution, instructions write back to the register files. Commitment (or retirement) is done in order for each thread out of the re-order buffers (`ReorderBuffer`s). Separate reorder buffers are maintained per thread.

3.2 Memory Hierarchy

3.2.1 Internal Cache Structure and Cache Coherence

A two-level cache hierarchy is modeled for the moment. Cache coherence is enforced with the directory-based MESI protocol between the private level one caches owned by each core and the level two cache that shared among cores.

Besides the data, tag, and state, a cache block has a corresponding directory entry that contains the owner and sharers information of the block. The geometry parameters of each cache in the memory hierarchy are configured via XML files. The cache subblock granularity is currently not supported but planned for future work. The LRU cache replacement policy is implemented for the moment, the implementations of more advanced policies are left as future work.

The details of the directory-based MESI protocols are described as below.

1. `findAndLock(addr, isBlocking, isRead, isRetry, onCompletedCallback)`.

```
uint set, way, tag;
MESIState state;

bool hit = this.cache.findBlock(addr, set, way, tag, state, true);

uint dumbTag;

if(!hit) {
  way = this.cache.replaceBlock(set);
  this.cache.getBlock(set, way, dumbTag, state);
}

DirLock dirLock = this.cache.dir.dirLocks[set];
if(!dirLock.lock()) {
  if(isBlocking) {
    onCompletedCallback(true, set, way, state, tag, dirLock);
  }
  else {
    this.retry({this.findAndLock(addr, isBlocking, isRead, true, onCompletedCallback);});
  }
}
else {
  this.cache[set][way].transientTag = tag;

  if(!hit && state != MESIState.INVALID) {
    this.schedule(
      {
        this.evict(set, way,
          (bool hasError)
          {
            uint dumbTag;

            if(!hasError) {
              this.stat.evictions++;
              this.cache.getBlock(set, way, dumbTag, state);
              onCompletedCallback(false, set, way, state, tag, dirLock);
            }
            else {
              this.cache.getBlock(set, way, dumbTag, state);
```

```
            dirLock.unlock();
            onCompletedCallback(true, set, way, state, tag, dirLock);
          }
        });
    }, this.hitLatency);

  }
  else {
    this.schedule(
      {
        onCompletedCallback(false, set, way, state, tag, dirLock);
      },
    this.hitLatency);
  }
}
```

2. load(addr, isRetry, onCompletedCallback).

```
this.findAndLock(addr, false, true, isRetry,
  (bool hasError, uint set, uint way, MESIState state, uint tag, DirLock dirLock)
  {
    if(!hasError) {
      if(!isReadHit(state)) {
        this.readRequest(this.next, tag,
        (bool hasError, bool isShared)
        {
          if(!hasError) {
            this.cache.setBlock(set, way, tag, isShared ? MESIState.SHARED
              : MESIState.EXCLUSIVE);
            this.cache.accessBlock(set, way);
            dirLock.unlock();
            onCompletedCallback();
          }
          else {
            dirLock.unlock();
            this.retry({this.load(addr, true, onCompletedCallback);});
          }
        });
      }
      else {
        this.cache.accessBlock(set, way);
        dirLock.unlock();
        onCompletedCallback();
      }
    }
    else {
      this.retry({this.load(addr, true, onCompletedCallback);});
    }
  });
```

3. store(addr, isRetry, onCompletedCallback).

```
this.findAndLock(addr, false, false, isRetry,
  (bool hasError, uint set, uint way, MESIState state, uint tag, DirLock dirLock)
  {
    if(!hasError) {
      if(!isWriteHit(state)) {
        this.writeRequest(this.next, tag,
          (bool hasError)
          {
            if(!hasError) {
              this.cache.accessBlock(set, way);
              this.cache.setBlock(set, way, tag, MESIState.MODIFIED);
              dirLock.unlock();
              onCompletedCallback();
            }
            else {
              dirLock.unlock();
              this.retry({this.store(addr, true, onCompletedCallback);});
            }
```

```
          });
      }
      else {
        this.cache.accessBlock(set, way);
        this.cache.setBlock(set, way, tag, MESIState.MODIFIED);
        dirLock.unlock();
        onCompletedCallback();
      }
    }
    else {
      this.retry({this.store(addr, true, onCompletedCallback);});
    }
  });
```

4. `evict(set, way, onCompletedCallback)`.
   Constant-latency (2) on-chip interconnect is assumed here.

```
uint tag;
MESIState state;

this.cache.getBlock(set, way, tag, state);

uint srcSet = set;
uint srcWay = way;
uint srcTag = tag;
CoherentCacheNode target = this.next;

this.invalidate(null, set, way,
  {
    if(state == MESIState.INVALID) {
      onCompletedCallback(false);
    }
    else if(state == MESIState.MODIFIED) {
      this.schedule(
        {
          target.evictReceive(this, srcTag, true,
            (bool hasError)
            {
              this.schedule(
                {
                  this.evictReplyReceive(hasError, srcSet, srcWay, onCompletedCallback);
                }, 2);
            });
        }, 2);
    }
    else {
      this.schedule(
        {
          target.evictReceive(this, srcTag, false,
            (bool hasError)
            {
              this.schedule(
                {
                  this.evictReplyReceive(hasError, srcSet, srcWay, onCompletedCallback);
                }, 2);
            });
        }, 2);
    }
  });
```

`evictReceive(source, addr, isWriteback, onReceivedReplyCallback)`.

```
this.findAndLock(addr, false, false, false,
      (bool hasError, uint set, uint way, MESIState state, uint tag, DirLock dirLock)
      {
        if(!hasError) {
          if(!isWriteback) {
            this.evictProcess(source, set, way, dirLock, onReceiveReplyCallback);
          }
```

```
          else {
            this.invalidate(source, set, way,
              {
                if(state == MESIState.SHARED) {
                  this.writeRequest(this.next, tag,
                    (bool hasError)
                    {
                      this.evictWritebackFinish(
                        source, hasError, set, way, tag, dirLock, onReceiveReplyCallback);
                    });
                }
                else {
                  this.evictWritebackFinish(
                    source, false, set, way, tag, dirLock, onReceiveReplyCallback);
                }
              });
          }
        }
        else {
          onReceiveReplyCallback(true);
        }
      });
```

evictWritebackFinish(source, hasError, set, way, tag, dirLock, onReceivedReplyCallback).

```
if(!hasError) {
  this.cache.setBlock(set, way, tag, MESIState.MODIFIED);
  this.cache.accessBlock(set, way);
  this.evictProcess(source, set, way, dirLock, onReceiveReplyCallback);
}
else {
  dirLock.unlock();
  onReceiveReplyCallback(true);
}
```

evictProcess(source, set, way, dirLock, onReceivedReplyCallback).

```
DirEntry dirEntry = this.cache.dir.dirEntries[set][way];
dirEntry.unsetSharer(source);
if(dirEntry.owner == source) {
  dirEntry.owner = null;
}
dirLock.unlock();
onReceiveReplyCallback(false);
```

evictReplyReceive(hasError, srcSet, srcWay, onCompletedCallback).

```
this.schedule(
  {
    if(!hasError) {
      this.cache.setBlock(srcSet, srcWay, 0, MESIState.INVALID);
    }
    onCompletedCallback(hasError);
  }, 2);
```

5. readRequest(target, addr, onCompletedCallback).
   Cconstant-latency (2) on-chip interconnect is assumed here.

```
this.schedule(
  {
    target.readRequestReceive(this, addr, onCompletedCallback);
  }, 2);
```

readRequestReceive(source, addr, onCompletedCallback).

```
this.findAndLock(addr, this.next == source, true, false,
  (bool hasError, uint set, uint way, MESIState state, uint tag, DirLock dirLock)
  {
    if(!hasError) {
      if(source.next == this) {
        this.readRequestUpdown(source, set, way, tag, state, dirLock, onCompletedCallback);
      }
      else {
        this.readRequestDownup(set, way, tag, dirLock, onCompletedCallback);
      }
    }
    else {
      this.schedule(
        {
          onCompletedCallback(true, false);
        }, 2);
    }
  });
```

readRequestUpdown(source, set, way, tag, state, dirLock, onCompletedCallback).

```
uint pending = 1;

if(state != MESIState.INVALID) {
  DirEntry dirEntry = this.cache.dir.dirEntries[set][way];

  if(dirEntry.owner !is null && dirEntry.owner != source) {
    pending++;
    this.readRequest(dirEntry.owner, tag,
      (bool hasError, bool isShared)
      {
        this.readRequestUpdownFinish(source, set, way, dirLock, pending, onCompletedCallback);
      });
  }

  this.readRequestUpdownFinish(source, set, way, dirLock, pending, onCompletedCallback);
}
else {
  this.readRequest(this.next, tag,
    (bool hasError, bool isShared)
    {
      if(!hasError) {
        this.cache.setBlock(set, way, tag, isShared ? MESIState.SHARED : MESIState.EXCLUSIVE);
        this.readRequestUpdownFinish(source, set, way, dirLock, pending, onCompletedCallback);
      }
      else {
        dirLock.unlock();
        this.schedule(
          {
            onCompletedCallback(true, false);
          }, 2);
      }
    });
}
```

readRequestUpdownFinish(source, set, way, dirLock, ref pending, onCompletedCallback).

```
pending--;
if(pending == 0) {
  DirEntry dirEntry = this.cache.dir.dirEntries[set][way];
  if(dirEntry.owner !is null && dirEntry.owner != source) {
    dirEntry.owner = null;
  }

  dirEntry.setSharer(source);
  if(!dirEntry.isShared) {
    dirEntry.owner = source;
  }
```

```
  this.cache.accessBlock(set, way);
  dirLock.unlock();
  this.schedule(
    {
      onCompletedCallback(false, dirEntry.isShared);
    }, 2);
}
```

readRequestDownup(set, way, tag, dirLock, onCompletedCallback).

```
uint pending = 1;

DirEntry dirEntry = this.cache.dir.dirEntries[set][way];
if(dirEntry.owner !is null) {
  pending++;
  this.readRequest(dirEntry.owner, tag,
    (bool hasError, bool isShared)
    {
      this.readRequestDownUpFinish(set, way, tag, dirLock, pending, onCompletedCallback);
    });
}

this.readRequestDownUpFinish(set, way, tag, dirLock, pending, onCompletedCallback);
```

readRequestDownUpFinish(set, way, tag, dirLock, ref pending, onCompletedCallback).

```
pending--;

if(pending == 0) {
  DirEntry dirEntry = this.cache.dir.dirEntries[set][way];
  dirEntry.owner = null;

  this.cache.setBlock(set, way, tag, MESIState.SHARED);
  this.cache.accessBlock(set, way);
  dirLock.unlock();
  this.schedule(
    {
      onCompletedCallback(false, false);
    }, 2);
}
```

6. writeRequest(target, addr, onCompletedCallback).

```
this.schedule(
  {
    target.writeRequestReceive(this, addr, onCompletedCallback);
  }, 2);
```

writeRequestReceive(source, addr, onCompletedCallback).

```
this.findAndLock(addr, this.next == source, false, false,
  (bool hasError, uint set, uint way, MESIState state, uint tag, DirLock dirLock)
  {
    if(!hasError) {
      this.invalidate(source, set, way,
        {
          if(source.next == this) {
            if(state == MESIState.MODIFIED || state == MESIState.EXCLUSIVE) {
              writeRequestUpdownFinish(
                source, false, set, way, tag, state, dirLock, onCompletedCallback);
            }
            else {
              this.writeRequest(this.next, tag,
                (bool hasError)
                {
```

```
                writeRequestUpdownFinish(
                  source, hasError, set, way, tag, state, dirLock, onCompletedCallback);
              });
          }
        }
        else {
          this.cache.setBlock(set, way, 0, MESIState.INVALID);
          dirLock.unlock();
          this.schedule(
            {
              onCompletedCallback(false);
            }, 2);
        }
      });
  }
  else {
    this.schedule(
      {
        onCompletedCallback(true);
      }, 2);
  }
});
```

writeRequestUpdownFinish(source, hasError, set, way, tag, state, dirLock, onCompletedCallback).

```
if(!hasError) {
  DirEntry dirEntry = this.cache.dir.dirEntries[set][way];
  dirEntry.setSharer(source);
  dirEntry.owner = source;

  this.cache.accessBlock(set, way);
  if(state != MESIState.MODIFIED) {
    this.cache.setBlock(set, way, tag, MESIState.EXCLUSIVE);
  }

  dirLock.unlock();
  this.schedule(
    {
      onCompletedCallback(false);
    }, 2);
}
else {
  dirLock.unlock();
  this.schedule(
    {
      onCompletedCallback(true);
    }, 2);
}
```

7. invalidate(except, set, way, onCompletedCallback).

```
uint tag;
    MESIState state;

    this.cache.getBlock(set, way, tag, state);

    uint pending = 1;

    DirEntry dirEntry = this.cache.dir.dirEntries[set][way];

    CoherentCacheNode[] sharersToRemove;

    foreach(sharer; dirEntry.sharers) {
      if(sharer != except) {
        sharersToRemove ~= sharer;
      }
    }

    foreach(sharer; sharersToRemove) {
```

```
      dirEntry.unsetSharer(sharer);
      if(dirEntry.owner == sharer) {
        dirEntry.owner = null;
      }

      this.writeRequest(sharer, tag,
        (bool hasError)
        {
          pending--;

          if(pending == 0) {
            onCompletedCallback();
          }
        });
      pending++;
    }

    pending--;

    if(pending == 0) {
      onCompletedCallback();
    }
```

3.2.2 On-Chip Interconnect

Currently, constant-latency on-chip interconnect is modeled, the cycle-accurate simulation of on-chip interconnect is left as future work.

3.2.3 Interface to External DRAM Simulators

Currently, constant-latency DRAM access is modeled, the cycle-accurate simulation of on-chip interconnect is left as future work. The integration of DRAMSim with Flexim is planned.

Chapter 4

Supporting Infrastructure

There are various supporting modules aside the aforementioned main components to maintain the Flexim's reusability.

4.1 Eventing and Callback Mechanisms

In flexim, cycle-accurate simulation is driven by event signals per cycle. Generally speaking, an event can be any piece of code that is scheduled to execute at the specified time. This piece of code can be represented as delegates in the D language. The delegate-based event queue (`DelegateEventQueue`) is illustrated as below:

```
class DelegateEventQueue: EventProcessor {
  this() {
  }

  void processEvents() {
    if(currentCycle in this.events) {
      foreach(event; this.events[currentCycle]) {
        event();
      }
      this.events.remove(currentCycle);
    }
  }

  void schedule(void delegate() event, ulong delay = 0) {
    this.events[currentCycle + delay] ~= event;
  }

  void delegate()[][ulong] events;
}
```

The delegate-based callback mechanism is used extensively in the implementation of the MESI cache coherence protocol. This way, the code is clean and readable.

4.2 Categorized Logging Mechanism

The logging component supports configurable logging functionalities that can facilitate development and even be useful after release. The code skeleton of the Logger class is shown as below:

```
class Logger {
  static this() {
    singleInstance = new Logger();
  }

  bool enabled(LogCategory category) {
    return category in this.logSwitches && this.logSwitches[category];
  }

  string message(string caption, string text) {
    return format("[%d] \t%s%s", currentCycle,
```

```
      caption.endsWith("info") ? "" : "[" ~ caption ~ "]␣", text);
  }

  void infof(LogCategory, T...)(LogCategory category, T args) {
    debug {
      this.info(category, format(args));
    }
  }

  void info(LogCategory category, string text) {
    debug {
      if(this.enabled(category)) {
        stdout.writeln(this.message(category ~ "|" ~ "info", text));
      }
    }
  }
...

  bool[LogCategory] logSwitches;

  static Logger singleInstance;
}
```

4.3 XML-Based Input/Output for Configurations and Statistics

Extensible Markup Language (XML) is a standard and pervasive mechanism for representing structural data in a machine-readable and human-friendly way. Here XML-based configuration specification and statistics output are provided. For example, the following code shows how to serialize and deserialize a cache configuration using the provided XML I/O support:

```
class CacheConfigXMLSerializer: XMLSerializer!(CacheConfig) {
  override XMLConfig save(CacheConfig cacheConfig) {
    XMLConfig xmlConfig = new XMLConfig("CacheConfig");

    xmlConfig["name"] = cacheConfig.name;
    xmlConfig["level"] = to!(string)(cacheConfig.level);
    xmlConfig["numSets"] = to!(string)(cacheConfig.numSets);
    xmlConfig["assoc"] = to!(string)(cacheConfig.assoc);
    xmlConfig["blockSize"] = to!(string)(cacheConfig.blockSize);
    xmlConfig["hitLatency"] = to!(string)(cacheConfig.hitLatency);
    xmlConfig["missLatency"] = to!(string)(cacheConfig.missLatency);
    xmlConfig["policy"] = to!(string)(cacheConfig.policy);

    return xmlConfig;
  }

  override CacheConfig load(XMLConfig xmlConfig) {
    string name = xmlConfig["name"];
    uint level = to!(uint)(xmlConfig["level"]);
    uint numSets = to!(uint)(xmlConfig["numSets"]);
    uint assoc = to!(uint)(xmlConfig["assoc"]);
    uint blockSize = to!(uint)(xmlConfig["blockSize"]);
    uint hitLatency = to!(uint)(xmlConfig["hitLatency"]);
    uint missLatency = to!(uint)(xmlConfig["missLatency"]);
    CacheReplacementPolicy policy = cast(CacheReplacementPolicy) (
      xmlConfig["policy"]);

    CacheConfig cacheConfig = new CacheConfig(
      name, level, numSets, assoc, blockSize, hitLatency, missLatency, policy);

    return cacheConfig;
  }

  static this() {
    singleInstance = new CacheConfigXMLSerializer();
  }

  static CacheConfigXMLSerializer singleInstance;
```

```
}
```

4.4 Plotting and Table Generation for Experiments

(Pre-release, Documentation-in-Progress)

Chapter 5

Evaluation, Limitations and Future Work

(Pre-release, Documentation-in-Progress)

5.1 Benchmark Evaluation

(Pre-release, Documentation-in-Progress)

5.1.1 Criteria

(Pre-release, Documentation-in-Progress)

5.1.2 Results

(Pre-release, Documentation-in-Progress)

5.2 Comparison to Other Simulators

(Pre-release, Documentation-in-Progress)

5.2.1 Results

(Pre-release, Documentation-in-Progress)

5.3 Limitations and Future Work

Here are some of the limitations of Flexim. This list is not intended to be comprehensive, but rather aims to provide the user with an initial understanding of the capabilities of the Flexim simulation environment. These capabilities are subject to change in future releases.

- Only statically compiled MIPS32 binaries are suported by Flexim. Multi-ISA support and loading support of dynamically compiled programs are remained as future work.
- The explicit modeling of Miss Status Holding Registers (MSHRs) is not implemented and left as future work.
- Cycle-accurate on-chip interconnect and dram controller modeling is not implemented and left as future work.