Appendix to Flexim Technical Report

1 Installation Instructions

1.1 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.

1.2 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.

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	OK
	Set-associative cache structure	OK
	Cache coherence	OK
	On-chip interconnect	Fixed-latency; Perf. Sim. Planned
	DRAM controller	Fixed-latency; Perf. Sim. 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

3 Implemented MIPS32 Instructions

3.1 Basic Instructions

1. `nop` (No operation). It does nothing.
2. `syscall` (Sysctem call). Implementation:

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

3. `sll` (Shift Word Left Logical). Meaning: $R_D = R_s \ll_{SA} 5$. Implementation:

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

4. `sllv` (Shift Word Left Logical Variable). Meaning: $R_D = R_S \ll R_{T_{4:0}}$. Implementation:

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

5. `sra` (Shift Word Right Arithmetic). Meaning: $R_D = R_S^{\pm} \gg_{SA} 5$. Implementation:

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

6. `srav` (Shift Word Right Arithmetic Variable). Meaning: $R_D = R_s^{\pm} \gg R_{T_{4:0}}$. Implementation:

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

7. `srl` (Shift Word Right Logical). Meaning: $R_D = R_s^{\varnothing} \gg_{SA} 5$. Implementation:

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

8. `srlv` (Shift Word Right Logical Variable). Meaning: $R_D = R_s^{\varnothing} \gg R_{T_{4:0}}$. Implementation:

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

3.2 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` (Branch). Meaning: $PC += _{OFF} 18^{\pm}$. Implementation:

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

3. `bal` (Branch and Link). Meaning: $R_A = PC + 8,\ PC += _{OFF} 18^{\pm}$. Implementation:

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

4. `beq` (Branch on Equal). Meaning: $_{IF}\ R_s = R_T,\ PC += _{OFF} 18^{\pm}$. Implementation:

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

5. `beqz` (Branch on Equal to Zero). Meaning: $_{IF}\ R_S = 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

6. `bgez` (Branch on Greater Than or Equal to Zero). Meaning: $_{IF}\ R_S \geq 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

7. `bgezal` (Branch on Greater Than or Equal to Zero and Link). Meaning: $R_A = PC + 8;\ _{IF}\ R_S \geq 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

8. `bgtz` (Branch on Greater Than Zero). Meaning: $IF\ R_S > 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

9. `blez` (Branch on Less Than or Equal to Zero). Meaning: $IF\ R_S \leq 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

10. `bltz` (Branch on Less Than Zero). Meaning: $IF\ R_S < 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

11. `bltzal` (Branch on Less Than Zero and Link). Meaning: $R_A = PC + 8;\ _{IF}\ R_S < 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

12. `bne` (Branch on Not Equal). Meaning: $_{IF}\ R_S \neq R_T,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

13. `bnez`(Branch on Not Equal to Zero). Meaning: $_{IF}\ R_S \neq 0,\ PC+ =_{OFF} 18^{\pm}$. Implementation:

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

14. `bc1f` (Branch on FP False). Meaning: . Implementation:

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

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

15. `bc1t` (Branch on FP True). Meaning: . Implementation:

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

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

16. `bc1fl` (Branch on FP False Likely). Meaning: . Implementation:

```
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` (Branch on FP True Likely). Meaning: . Implementation:

```
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;
}
```

3.3 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` (Jump). Meaning: $PC = PC_{31:28} ::_{ADDR} 28^{\emptyset}$.
   Its target PC calculation is shown as below:

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

   Implementation:

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

3. `jal` (Jump and Link). Meaning: $R_A = PC + 8$; $PC = PC_{31:28} ::_{ADDR} 28^{\emptyset}$.
   Its target PC calculation is shown as below:

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

   Implementation:

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

4. `jalr` (Jump and Link Register). Meaning: $R_D = PC + 8;\ PC = R_S$.
   Its target PC calculation is shown as below:

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

   Implementation:

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

5. `jr` (Jump Register). Meaning: $PC = R_S$.
   Its target PC calculation is shown as below:

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

   Implementation:

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

3.4 Floating Point Arithmetic Instructions

1. `add_d` (Add Double). Meaning: . Implementation:

   ```
   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` (Subtract Double). Meaning: . Implementation:

   ```
   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` (Multiply Double). Meaning: . Implementation:

   ```
   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` (Divide Double). Meaning: . Implementation:

   ```
   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` (Square Root Double). Meaning: . Implementation:

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

   double fd = sqrt(fs);

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

6. abs_d (Absolute Value Double). Meaning: . Implementation:

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

double fd = fabs(fs);

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

7. neg_d (Negate Double). Meaning: . Implementation:

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

double fd = -1 * fs;

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

8. mov_d (Move Register Double). Meaning: . Implementation:

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

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

9. add_s (Add Single). Meaning: . Implementation:

```
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 (Subtract Single). Meaning: . Implementation:

```
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 (Multiply Single). Meaning: . Implementation:

```
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 (Divide Single). Meaning: . Implementation:

```
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 (Square Root Single). Meaning: . Implementation:

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

float fd = sqrt(fs);

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

14. `abs_s` (Absolute Value Single). Meaning: . Implementation:

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

float fd = fabs(fs);

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

15. `neg_s` (Negate Single). Meaning: . Implementation:

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

float fd = -fs;

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

16. `mov_s` (Move Register Single). Meaning: . Implementation:

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

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

17. `cvt_d_s` (Convert from single floating point to double floating point). Meaning: . Implementation:

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

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

18. `cvt_w_s` (Convert from single floating point to 32-bit fixed point). Meaning: . Implementation:

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

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

19. `cvt_l_s` (Convert from single floating point to longword fixed point). Meaning: . Implementation:

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

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

20. `cvt_s_d` (Convert from double floating point to single floating point). Meaning: . Implementation:

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

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

21. `cvt_w_d` (Convert from double floating point to 32-bit fixed point). Meaning: . Implementation:

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

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

22. `cvt_l_d` (Convert from double floating point to longword fixed point). Meaning: . Implementation:

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

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

23. `cvt_s_w` (Convert from 32-bit fixed point to single floating point). Meaning: . Implementation:

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

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

24. `cvt_d_w` (Convert from 32-bit fixed point to double floating point). Meaning: . Implementation:

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

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

25. `cvt_s_l` (Convert from longword fixed point to single floating point). Meaning: . Implementation:

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

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

26. `cvt_d_l` (Convert from longword fixed point to double floating point). Meaning: . Implementation:

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

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

27. `c_<cond>_d` (Floating Point compare double) 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`. Meaning: . Implementation:

```
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` (Floating Point compare single) 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`. Meaning: . Implementation:

```
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` (Move Word From Floating Point). Meaning: . Implementation:

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

30. `cfc1` (Move Control Word From Floating Point). Meaning: . Implementation:

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

uint rt = 0;

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

31. `mtc1` (Move Word To Floating Point). Meaning: . Implementation:

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

32. `ctc1` (Move Control Word To Floating Point). Meaning: . Implementation:

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

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

3.5 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` (Add Word). Meaning: $R_D = R_s + R_T$. Overflow trap. Implementation:

```
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` (Add Immediate Word). Meaning: $R_D = R_S +_{const} 16^{\pm}$. Overflow trap. Implementation:

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

4. `addiu` (Add Immediate Unsigned Word). Meaning: $R_D = R_S +_{const} 16^{\pm}$. Implementation:

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

5. `addu` (Add Unsigned Word). Meaning: $R_D = R_s + R_T$. Implementation:

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

6. `sub` (Subtract Word). Meaning: $R_D = R_S - R_T$. Overflow trap. Implementation:

```
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` (Subtract Unsigned Word). Meaning: $R_D = R_S - R_T$. Implementation:

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

8. `and` (And). Meaning: $R_D = R_s \& R_T$. Implementation:

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

9. `andi` (And Immediate). Meaning: $R_D = R_S \&_{const} 16^{\varnothing}$. Implementation:

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

10. `nor` (Nor). Meaning: $R_D =\sim (R_S | R_T)$. Implementation:

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

11. `or` (Or). Meaning: $R_D = R_S | R_T$. Implementation:

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

12. `ori` (Or Immediate). Meaning: $R_D = R_S |_{const} 16^{\varnothing}$. Implementation:

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

13. `xor` (Exclusive Or). Meaning: $R_D = R_S \oplus R_T$. Implementation:

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

14. `xori` (Exclusive Or Immediate). Meaning: $R_D = R_S \oplus_{const} 16^{\varnothing}$. Implementation:

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

15. `slt` (Set on Less Than). Meaning: $R_D = (R_S^{\pm} < R_T^{\pm})?1:0$. Implementation:

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

16. `slti` (Set on Less Than Immediate). Meaning: $R_D = (R_S^{\pm} <_{const} 16^{\pm})?1:0$. Implementation:

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

17. `sltiu` (Set on Less Than Immediate Unsigned). Meaning: $R_D = (R_S^{\varnothing} <_{const} 16^{\varnothing})?1:0$. Implementation:

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

18. `sltu` (Set on Less Than Unsigned). Meaning: $R_D = (R_S^{\varnothing} < R_T^{\varnothing})?1:0$. Implementation:

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

19. `lui` (Load Upper Immediate). Meaning: $R_D =_{const} 16 \ll 16$. Implementation:

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

20. `divu` (Divide Unsigned Word). Meaning: $L_O = R_S^{\varnothing}/R_T^{\emptyset}$; $H_I = R_S^{\emptyset}\ mod\ R_T^{\emptyset}$. Implementation:

```
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` (Divide Word). Meaning: $L_O = R_S^{\pm}/R_T^{\pm}$; $H_I = R_S^{\pm}\ mod\ R_T^{\pm}$. Implementation:

```
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` (Move From LO). Meaning: $R_D = L_O$. Implementation:

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

23. `mfhi` (Move From HI). Meaning: $R_D = H_I$. Implementation:

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

24. `mtlo` (Move To LO). Meaning: $L_O = R_D$. Implementation:

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

25. `mthi` (Move To HI). Meaning: $H_I = R_D$. Implementation:

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

26. `mult` (Multiply Word). Meaning: $A_{CC} = R_S^{\pm} \times R_T^{\pm}$. Implementation:

```
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` (Multiply Unsigned Word). Meaning: $A_{CC} = R_S^{\emptyset} \times R_T^{\emptyset}$. Implementation:

```
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;
```

3.6 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` (Load Byte). Meaning: $R_T =_{MEM} 8(R_S +_{OFF} 16^{\pm})^{\pm}$. Implementation:

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

3. `lbu` (Load Byte Unsigned). Meaning: $R_T =_{MEM} 8(R_S +_{OFF} 16^{\pm})^{\emptyset}$. Implementation:

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

4. `lh` (Load Halfword). Meaning: $R_T =_{MEM} 16(R_S +_{OFF} 16^{\pm})^{\pm}$. Implementation:

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

5. `lhu` (Load Halfword Unsigned). Meaning: $R_T =_{MEM} 16(R_S +_{OFF} 16^{\pm})^{\varnothing}$. Implementation:

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

6. `lw` (Load Word). Meaning: $R_T =_{MEM} 32(R_S +_{OFF} 16^{\pm})$. Implementation:

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

7. `lwl` (Load Word Left).
   Its overriden effective address calculation is shown as below:

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

   Meaning: $R_T = LoadWordLeft(R_S +_{OFF} 16^{\pm})$. Implementation:

```
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` (Load Word Right).
   Its overriden effective address calculation is shown as below:

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

   Meaning: $R_T = LoadWordRight(R_S +_{OFF} 16^{\pm})$. Implementation:

```
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` (Load Linked Word). Meaning: $R_T =_{MEM} 32(R_S +_{OFF} 16^{\pm}); LINK$. Implementation:

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

10. `lwc1` (Load Word to Coprocessor-1). Meaning: . Implementation:

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

11. `ldc1` (Load Doubleword to Coprocessor-1). Meaning: . Implementation:

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

12. `sb` (Store Byte). Meaning: ${}_{MEM}8(R_T +_{OFF} 16^{\pm}) = R_{S_{7:0}}$. Implementation:

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

13. `sh` (Store Halfword). Meaning: ${}_{MEM}16(R_T +_{OFF} 16^{\pm}) = R_{S_{15:0}}$. Implementation:

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

14. `sw` (Store Word). Meaning: ${}_{MEM}32(R_T +_{OFF} 16^{\pm}) = R_S$. Implementation:

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

15. `swl` (Store Word Left).
Its overriden effective address calculation is shown as below:

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

Meaning: $StoreWordLeft(R_T +_{OFF} 16^{\pm}, R_S)$. Implementation:

```
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` (Store Word Right).
Its overriden effective address calculation is shown as below:

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

Meaning: $StoreWordRight(R_T +_{OFF} 16^{\pm}, R_S)$. Implementation:

```
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` (Store Conditional Word). Meaning: $_{MEM}32(R_S +_{OFF} 16^{\pm}) = R_T$; $R_T = 1$. Implementation:

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

18. `swc1` (Store Word from Coprocessor-1). Meaning: . Implementation:

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

19. `sdc1` (Store Doubleword from Coprocessor-1). Meaning: . Implementation:

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

4 Implemented System Calls

1. exit. Syscall number: 1. Meaning: terminate the current process. Implementation:

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

2. read. Syscall number: 3; Meaning: read from a file descriptor. Implementation:

```
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. Syscall number: 4; Meaning: write to a file descriptor. Implementation:

```
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. Syscall number: 5, Meaning: open a file or device. Implementation:

```
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. Syscall number: 6; Meaning: close a file descriptor. Implementation:

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

6. lseek. Syscall number: 19; Meaning: reposition read/write file offset. Implementation:

```
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. Syscall number: 20. Meaning: get process identification. Implementation:

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

8. getuid. Syscall number: 24. Meaning: get real user ID. Implementation:

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

9. brk. Syscall number: 45. Meaning: change the amount of space allocated for the calling process's data segment. Implementation:

```
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. Syscall number: 47. Meaning: get real group ID. Implementation:

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

11. geteuid. Syscall number: 49. Meaning: get effective user ID. Implementation:

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

12. getegid. Syscall number: 50. Meaning: get effective group ID. Implementation:

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

13. fstat. Syscall number: 28. Meaning: get file status. Implementation:

```
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. Syscall number: 122. Meaning: get name and information about current Linux kernel. Implementation:

```
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. Syscall number: 140. Meaning: move extended read/write file pointer. Implementation:

```
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;
```

5 Implementation of Key OoO Pipeline Structures and Pipeline Stages

5.1 Structures

(Pre-release, Documentation-in-Progress)

5.2 Pipeline Stages

(Pre-release, Documentation-in-Progress)

6 Implementation of the directory-based MESI Cache Coherence Protocol

The implementation of the directory-based MESI protocols are as follows.

1.
```
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();
    }
```