virtutech

Simics Micro-Architectural Interface

Simics Version	3.0
Revision	1406
Date	2008-02-19

Contents

Chapter 1

About Simics Documentation

1.1 Conventions

Let us take a quick look at the conventions used throughout the Simics documentation. Scripts, screen dumps and code fragments are presented in a `monospace` font. In screen dumps, user input is always presented in bold font, as in:

```
Welcome to the Simics prompt
simics> this is something that you should type
```

Sometimes, artificial line breaks may be introduced to prevent the text from being too wide. When such a break occurs, it is indicated by a small arrow pointing down, showing that the interrupted text continues on the next line:

```
This is an artificial ↵
line break that shouldn't be there.
```

The directory where Simics is installed is referred to as `[simics]`, for example when mentioning the `[simics]/README` file. In the same way, the shortcut `[workspace]` is used to point at the user's workspace directory.

1.2 Simics Guides and Manuals

Simics comes with several guides and manuals, which will be briefly described here. All documentation can be found in `[simics]/doc` as Windows Help files (on Windows), HTML files (on Unix) and PDF files (on both platforms). The new Eclipse-based interface also includes Simics documentation in its own help system.

Simics Installation Guide for Unix and for Windows

These guides describe how to install Simics and provide a short description of an installed Simics package. They also cover the additional steps needed for certain features of Simics to work (connection to real network, building new Simics modules, . . .).

Simics User Guide for Unix and for Windows

These guides focus on getting a new user up to speed with Simics, providing information on Simics features such as debugging, profiling, networks, machine configuration and scripting.

Simics Eclipse User Guide

This is an alternative User Guide describing Simics and its new Eclipse-based graphical user interface.

Simics Target Guides

These guides provide more specific information on the different architectures simulated by Simics and the example machines that are provided. They explain how the machine configurations are built and how they can be changed, as well as how to install new operating systems. They also list potential limitations of the models.

Simics Programming Guide

This guide explains how to extend Simics by creating new devices and new commands. It gives a broad overview of how to work with modules and how to develop new classes and objects that fit in the Simics environment. It is only available when the DML add-on package has been installed.

DML Tutorial

This tutorial will give you a gentle and practical introduction to the Device Modeling Language (DML), guiding you through the creation of a simple device. It is only available when the DML add-on package has been installed.

DML Reference Manual

This manual provides a complete reference of DML used for developing new devices with Simics. It is only available when the DML add-on package has been installed.

Simics Reference Manual

This manual provides complete information on all commands, modules, classes and haps implemented by Simics as well as the functions and data types defined in the Simics API.

Simics Micro-Architectural Interface

This guide describes the cycle-accurate extensions of Simics (Micro-Architecture Interface or MAI) and provides information on how to write your own processor timing models. It is only available when the DML add-on package has been installed.

RELEASENOTES and LIMITATIONS files

These files are located in Simics's main directory (i.e., `[simics]`). They list limitations, changes and improvements on a per-version basis. They are the best source of information on new functionalities and specific bug fixes.

Simics Technical FAQ

This document is available on the Virtutech website at `http://www.simics.net/support`. It answers many questions that come up regularly on the support forums.

Simics Support Forum

The Simics Support Forum is the main support tool for Simics. You can access it at `http://www.simics.net`.

Other Interesting Documents

Simics uses Python as its main script language. A Python tutorial is available at `http://www.python.org/doc/2.4/tut/tut.html`. The complete Python documentation is located at `http://www.python.org/doc/2.4/`.

Chapter 2

Introduction

This document describes how to model cycle accurate microprocessors and memory systems with Simics. A basic knowledge of Simics and its time modeling (refer to *Understanding Simics Timing* in *Simics User Guide*) is recommended before reading on.

Simics is a system-level instruction set simulator; its default CPU models are functionally very close to their real counterparts, detailed enough to boot and run unmodified operating systems and applications. At this level of abstraction, exact timing is rarely a requirement. Simics considers the execution of an entire instruction, an exception or an interrupt as an atomic operation in the simulation process: it takes exactly one cycle.

Simics allows a user module to take control over the memory system timing, deciding how many cycles each memory transaction requires to complete. Although this model can be sufficient to simulate the effects of caches, it still enforces Simics's concept of atomic, in-order execution.

Simics Micro Architectural Interface (MAI) was designed to overcome these limitations while keeping the power of a functional full-system simulator. Using MAI, Simics can model the timing behavior of modern processors with deep pipelines and still run unmodified system-level software.

The basic idea behind MAI is to let the user decide when things happen, while Simics handles how things happen. A user module chooses when to fetch, decode, execute and commit instructions, using MAI to tell Simics to actually perform the actions. Execution is supervised by Simics; it will notify the user when program order consistency is violated, but even these warnings may be overridden.

Simics MAI supports out-of-order execution, multi-processor and multi-threading, branch and value speculation; for each processor, it gives the user control over an execution tree that represents the possible execution paths. The user module builds these paths by speculating on output values and branches.

Simics MAI is available for the SPARC-V9 and x86 architectures. The SPARC-V9 architecture, introduced in Simics 1.4, is more mature than the x86 implementation delivered with Simics 2.0, but both support all MAI features.

Note: The Parameterized execution mode that was present up to Simics 1.8 (for SPARC only) is not available any more. The **ooo-micro-arch** module mimics its behavior and should be used instead.

Chapter 3

Simics MAI Execution Model

This section provides an overview of Simics MAI concepts.

3.1 Dependences

An out-of-order CPU core needs to keep track of all dependences in the instruction stream so that instructions do not get executed in the wrong order. We describe below the different types of dependences and how Simics handles them.

3.1.1 Register Dependences

Figure 3.1 shows three types of dependences, read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW). We are using assembly code with the format *opcode source1, source2, destination*.

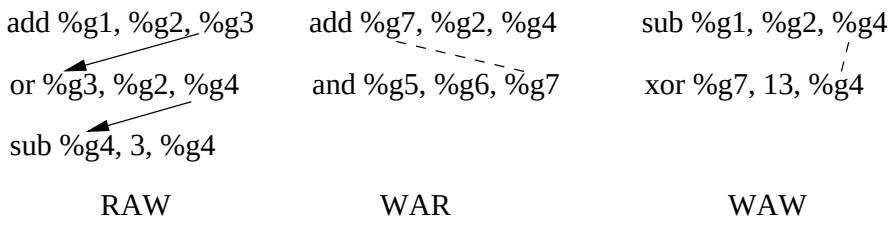

Figure 3.1: Different types of dependences

RAW dependences are also called *data dependences* since there is a flow of data between the dependent instructions. A data dependence exists between instruction x and y if y uses a result directly produced by x or produced by an instruction that has a RAW dependence on x. Instructions with RAW dependences can not be reordered. Simics will ensure that RAW dependences are followed and will refuse to execute instructions that did not receive all the input they needed (although the user may also speculate on the input values, thus fulfilling this requirement).

WAR and WAW dependences are sometimes called *anti-dependences* or *name dependences* since the two instructions use the same resource but there is no flow of data between them. A name dependence exists if:

- Instruction x reads a register or memory location that instruction y writes to or
- Instruction x writes to a register or memory location that instruction y also writes to.

Instruction WAW and WAR dependences can be reordered by allocating more resources to their execution. In the WAW case, the CPU could write to an internal register %r0 instead of %g4, thus removing the resource conflict and the dependence. This is commonly called register renaming. Simics has the capability to handle all anti-dependences by using an unlimited pool of internal registers.

SPARC-V9

Simics MAI for SPARC-V9 can rename the following registers:

- All general purpose registers, i.e. %g1 - %g7, %o0 - %o7, %l0- %l7, and %i0 - %i7. %g0 does not introduce any dependence since it always reads as zero.
- The register window state registers, i.e. %cwp, %cansave, %canrestore, %otherwin, and %cleanwin.
- The integer condition code register %ccr.
- The multiply and divide register %y.
- All floating-point registers, %f0 - %f63. They are treated as single entities, e.g. if an instruction is using a quad register for input it will depend on four separate single precision registers.
- The fcc0, fcc1, fcc2, fcc3, ftt, aexc, and cexc parts of the floating-point state register %fsr.
- The floating point register state register %fprs.
- The graphic status register %gsr.
- The alternate space identifier register %asi.

Instructions accessing other registers need to be synchronized and cannot run out of order. See section 4.5 for more information.

x86

Simics MAI for x86 can rename the following registers:

- EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI.
- MMX registers.

- XMM registers.
- EFlags register.

Instructions accessing other registers need to be synchronized and cannot run out of order. See section 4.5 for more information.

Note that the list does not include the x87 FPU that uses a stack model to represent floating-point registers. All x87 FPU instructions are currently synchronized.

3.1.2 Control Dependences

```
0xfefc  cmp %i3,%i4
0xff00  beq 0xff0c
0xff04  nop
0xff08  add %g2,%g3,%g4
0xff0c  ...
```

Figure 3.2: A control dependence

Figure 3.2 is an example of control dependence: a conditional branch instruction before an add instruction. The add cannot be executed unless the branch target is known. Control dependences can be avoided by speculating on the execution path. In Simics MAI control dependences are handled by speculating on the Program Counter register (PC or IP depending on the architecture). Instructions that have been fetched with a speculative PC are marked as "speculative". The speculative status will be removed as soon as the control path is known. If a speculative instruction was incorrectly executed, it has to be squashed before the execution can continue.

3.1.3 Memory Dependences

Memory data dependences are handled by an internal load-store queue (LSQ) and an optional consistency controller. The LSQ handles self-consistency (i.e. ensures that program order consistency is fulfilled) and the consistency controller handles memory consistency for multiprocessor systems.

When a store is executed, the transaction is placed in a store queue without being issued to memory. When the store instruction becomes non-speculative, the store operation can be retired and sent to memory. Speculative loads are matched against the LSQ before being sent to memory to ensure that they load the current (and possibly speculative) value of the memory area they access.

The LSQ will also make sure that stores are never committed out of order if they conflict and that loads cannot execute if there is a risk of conflict with a non-executed previous store.

The size of the LSQ is unlimited. It is up to the user module to place restrictions on the number of outstanding stores.

The consistency controller is described in section 5.3.

3.2 Instruction Tree

Simics builds a tree of active instructions to keep track of dependence information. As instructions enter the out-of-order window, they are added as leaves to the tree with the previous instruction in program order as their parent. Instructions can only be committed from the window when they are at the root of the tree. Branches occur at speculation points. Currently two forms of speculation are supported, control speculation and data speculation.

Control speculation occurs when the user model generates a fetch address that has not yet been produced by an earlier correct instruction. The instruction being fetched will be regarded as speculative but not necessarily incorrect . As soon as the previous instruction executes non-speculatively the address will be compared against the speculative address. If there is a match the instruction will be considered correct and no longer speculative.

Data speculation occurs when the user supplies an input value to an instruction that has not yet been generated by an earlier correct instruction. As with fetches the instruction will be speculative until the correct value arrives. If the value does not match, the instruction becomes incorrect and need to be rewinded and re-executed or squashed.

If the control or data speculation generates an address or value that has already been produced by an earlier correct instruction the comparison occurs immediately. Again, for matches the instruction will be correct and for mismatches the instruction will be speculative.

To keep track of speculation, Simics labels each instruction as speculative or correct. Speculative does not imply incorrect, unless a correct branch exists. This structure allows multiple speculative instruction paths to be active. In the general case, to know if an instruction is correct or incorrect, all previous instructions need to be executed and non-speculative.

While Simics and the underlying MAI support an unlimited number of speculative execution paths, the current way it is used can add restrictions. The MAI is limited only by the aggressiveness of the user model. The model can inspect the state of a branch, deciding when incorrect instruction should be squashed.

The maximum size of the instruction tree is set by an attribute called *reorder_buffer_size* in each CPU.

3.2.1 Tracking Instruction Execution

To keep track of the execution, each instruction can be queried on its current phase, its current status and a flag indicating whether the instruction is speculative or not. The phase describes where the instruction is in an idealized pipeline. While this might suggest an enormous number of possible states for each instruction, not all combinations are legal or meaningful.

The six phases of an instruction are:

Init
: A place for the instruction has been reserved in the out-of-order window. At this point, it is possible to speculatively set the fetch address.

Fetch
: The instruction has been fetched from memory. If Simics was configured to do so, an

instruction fetch transaction has been sent to the memory hierarchy and it has finished stalling. The instruction opcode is available.

Decode
The instruction has been decoded: its type and its input and output registers are known. It is possible to query and set the values of the input registers. If not all the input values are known the instruction cannot be executed. No memory transaction is performed during this phase so it can not stall.

Execute
The instruction has been executed: all output values have been produced. Loads have been completed; stores have been issued and are waiting in the LSQ. The instruction may still be speculative.

Retire
To enter this phase, the instruction has to be non-speculative. Stores are sent to memory from the LSQ.

Commit
To enter this phase, an instruction must be root of the tree. The instruction results are committed to the architectural register state.

An instruction is considered non-speculative when all the input values have been validated against the previous non-speculative instructions that produced them. The fetch address (PC) is included in the input values (control speculation).

The **Retire** and **Commit** phases could be considered as part of a global commit phase. The reason they are separated is to allow users to retire stores out-of-order. Simics requires however that commit to the architectural state be done in-order (when instructions are root of the tree).

The status of an instruction is one of:

Waiting
Some input is missing and no speculation is done.

Ready
All needed input have been produced or speculated.

Stalling
Executing a memory transaction. This can happen during the Fetch, Execute and Retire phases.

Faulting
The instruction has triggered an exception.

Trap (*x86 only*)
The instruction has triggered a trap.

Interrupt (*x86 only*)
An immediate interrupt must be taken before executing this instruction.

3.2.2 Instruction Tree Example

Here follows an example of an instruction tree using SPARC assembly:

```
 0 <0xf000e7b8> stx %g4, [%o2 + -16]           R
 1 <0xf000e7bc> cmp %g4, 0                     R
 2 <0xf000e7c0> bne,pt %xcc, 0xf000e7ac        E
     3 <0xf000e7c4> add %o2, 16, %o2               E
     4 <0xf000e7ac> ldx [%o2 + 8], %g4             D (stalling)
     5 <0xf000e7b0> stx %g4, [%o2 + -8]            D (waiting for %g4) S
     6 <0xf000e7b4> ldx [%o2 + 0], %g4             D (stalling)
     7 <0xf000e7b8> stx %g4, [%o2 + -16]           D (waiting for %g4) S
     8 <0xf000e7bc> cmp %g4, 0                     D (waiting for %g4) S
     9 <0xf000e7c0> bne,pt %xcc, 0xf000e7ac        D (waiting for %cc) S
 3 <0xf000e7c4> add %o2, 16, %o2               E S
 4 <0xf000e7c8> ld [%i4 + 0], %g4              D (stalling)
 5 <0xf000e7cc> add %g4, 1, %g4                D (waiting for %g4) S
 6 <0xf000e7d0> st %g4, [%i4 + 0]              D (waiting for %g4) S
 7 <0xf000e7d4> jmpl [%i7 + 8], %g0            E S
     8 <0xf000e7d8> restore %g0, %g0, %g0          E S
     9 <0xf0009c18> sethi %hi(0x4000), %i2         E S
    10 <0xf0009c1c> bne 0xf00096a8                 E S
        11 <0xf0009c20> nop                            E S
        12 <0xf00096a8> mov %i2, %o0                   E S
    11 <0xf0009c20> mov %i2, %o0                   E S
    12 <0xf0009c24> srl %o0, 0, %l0                E S
    13 <0xf0009c28> sethi %hi(0xf04b2400), %o0     E S
    14 <0xf0009c2c> add %o0, 896, %i0              E S
    15 <0xf0009c30> srl %i3, 0, %o1                E S
    16 <0xf0009c34> mov %i0, %o0                   E S
 8 <0xf000e7d8> restore %g0, %g0, %g0          E S
 9 <0xf000e7dc> bcs,a,pt %xcc, 0xf000e814      E S
10 <0xf000e7e4> ldx [%o2 + 0], %o0             D (ready) S
11 <0xf000e7e8> cmp %o0, 0                     F
12 <0xf000e7ec> be,a,pt %xcc, 0xf000e814       F
13 <0xf000e7f0> ld [%i4 + 0], %g4              F
```

The tree shows instructions in several different phases. Instructions 2, 7, and 10 are branch instructions that each have 2 predicted targets (jump target and fall through). This has created 4 possible paths through the tree. F stands for Fetched, D for Decoded, E for Executed, and R for retired. Committed instructions have been removed from the tree. S means that the instruction is marked as speculative.

Chapter 4

The Micro-Architectural Interface

The Micro-Architectural Interface allows the user module to specify all actions a processor should take during one clock cycle. This gives the user total control of the scheduling and latencies of the different instruction phases described earlier. The module is of course not limited to these phases alone, it has full freedom to add its own phases or pipeline stages. The simulation is driven forward by calling API functions that will pass instructions through the different phases. It is up to the user module to decide *when* things should happen while Simics will still perform *what* should happen. This allows the user to concentrate on the timing aspects of the processor while leaving the functional details to Simics.

This extensive API allows the user to control when instruction should be fetched, decoded, executed, etc. As described earlier, the active instruction state is stored in a tree. This helps the user keep track of dependences between instruction and prevents instructions from being executed before input data is available.

Complete details of all API functions are documented in appendix A.

4.1 Cycle Handler

The entry to the Micro Architectural Interface is a cycle handler. It is a call-back function that is called every clock cycle and is the location for the user code that models the micro architecture of the processor. The call-back should be placed in a module that can be loaded into Simics (see the *Simics User Guide* for a detailed description on how to write loadable modules).

The cycle handler can be configured to be called in different ways depending on the machine that will be modeled. If a single processor system or a SMP system is the target (where all the processors are independent) the cycle handler can be set up so that it is called once for each processor during a clock cycle, i.e. the whole first cycle is modeled for the first processor, then the whole first cycle is modeled for the second processor, etc. until all processors have modeled their first cycle, then the second cycle is modeled for the first processor and so on.

If a SMT system should be modeled each thread will be mapped on one Simics processor so that each physical processor will actually consists of a group of Simics processors. One cycle handler should then be used per group since the threads in one group will be dependent and may interact with each other during the same cycle.

An example of how to set up a cycle handler is presented below.

```
...

void
my_cycle_handler(conf_object_t *cpu, void *data)
{
    // All threads on one CPU is modeled here, cpu will
    // point the the first thread on each physical CPU.

    // when we are finished we post ourself in the next cycle
    SIM_time_post_cycle(cpu, 1, 0, my_cycle_handler, data);
}

static void
setup(void *data)
{
    /* We set up a multi-processor system with 3 CPUs.
       Each CPU consists of 2 threads. We need a 6 processor
       configuration for this to work where,
       cpu0 and cpu1 are threads on the first physical processor,
       cpu2 and cpu3 are threads on the second physical processor,
       cpu4 and cpu5 are threads on the third physical processor.
    */

    SIM_time_post_cycle(SIM_get_object("cpu0"),
                        0, 0, my_cycle_handler, data);
    SIM_time_post_cycle(SIM_get_object("cpu2"),
                        0, 0, my_cycle_handler, data);
    SIM_time_post_cycle(SIM_get_object("cpu4"),
                        0, 0, my_cycle_handler, data);
}

/* This code is called when the module is loaded into Simics */
DLL_EXPORT void
init_local(void)
{
    void *user_data = 0;

    /* We call the setup function as soon as we have a ready
       configuration */
    if (SIM_initial_configuration_ok())
        setup(user_data);
    else
        SIM_hap_register_callback("Core_Initial_Configuration",
```

```
                                    setup, user_data);
}

...
```

4.2 Creating the Instruction Tree

As stated previously, the state for an instruction in the out-of-order window is maintained in an Instruction Tree. Instructions in the tree are identified by an instruction id which is returned when the data structures for the instruction are allocated through the API function ***SIM_instruction_begin***. All API functions that operate on an instruction take the instruction id as parameter.

Once an instruction is created with ***SIM_instruction_begin*** it can be inserted into the instruction tree with ***SIM_instruction_insert***. This function takes the id of the instruction to insert and the id of a parent instruction. If the parent instruction already has a child the new instruction will be added as an additional child, thus creating a branch in the tree.

4.3 Proceeding Instructions

So far our instruction tree contains only empty instruction data structures. ***SIM_instruction_proceed*** is used both to initialize these data structures and to move the instruction between the phases of instruction operation. The defined instruction phases are: initiated, fetched, decoded, executed, retired, and committed. More phases may be added in the future. These phases do not map directly to a modern processor pipeline but define the atomic phases an instruction can be broken down to inside Simics. Somewhere along the pipeline an instruction need to pass through some or all of these phases.

The user can define which phases to proceed to by calling ***SIM_instruction_set_stop_phase***. This function takes a phase and a boolean value. ***SIM_instruction_proceed*** will then stop after the next phase that is set to true. For convenience there are predefined functions that combine setting a stop phase and proceeding. These shortcuts are: ***SIM_instruction_fetch***, ***SIM_instruction_decode***, ***SIM_instruction_execute***, ***SIM_instruction_retire***, and ***SIM_instruction_commit***.

Instructions can be proceeded to the decoded phase before they are inserted in the instruction tree, this is useful if the tree location for a decoded instruction depends on its type (the type of an instruction can be determined when it is decoded). For example the instruction following a delay slot have a different parent depending on whether the delay slot is annulled or not.

4.3.1 Fetch Phase

During the fetch phase an instruction is copied from memory into the internal instruction data structures created by ***SIM_instruction_insert***. As this is a memory operation, any memory hierarchy attached to Simics will be called during the phase. The memory

hierarchy may indicate that the memory access should stall the CPU. If this occurs ***SIM_instruction_proceed*** or its shortcuts will return a status value to notify the caller about this. ***SIM_instruction_remaining_stall_time*** will return the number of cycles remaining. Once the stall has completed ***SIM_instruction_remaining_stall_time*** will return zero and the instruction can be proceeded to the next phase. If ***SIM_instruction_proceed*** or any of its shortcuts is called before the stall time has completed they will return the same status value again.

Fetching an instruction may also generate an exception. In this case another status value is returned. Exception handling is described in section 4.3.6.

A program counter value is necessary to fetch an instruction. Simics can look for the value in several places. If the instruction is at the head of the instruction tree or not yet inserted in the tree, Simics will read the architectural state. For instructions at other locations in the tree, Simics will look in the output of the parent instruction. If the parent has not reached the executed phase the program counter output field will be invalid and the instruction will not be able to proceed to fetched. However the user can also override the default program counter value with the function ***SIM_instruction_write_input_reg***. This is the only way to fetch instructions whose parent has not yet executed. This will also mark the instruction as speculative. As soon as the parent instruction is executed the child will become non-speculative if the produced value of the program counter matches the one set (if the parent was non-speculative). When the written program counter does not match the generated program counter the instruction is incorrect and will eventually need to be discarded. See section 4.4.

When simulating the SPARC architecture the above applies to nPC as well.

When an instruction is fetched ***SIM_instruction_opcode*** can be used to retrieve the opcode.

4.3.2 Decode Phase

During the decode phase Simics translates the fetched instruction into the internal data structures that are used during execution. It is then possible to use ***SIM_instruction_type*** to determine the type of the instruction and ***SIM_instruction_get_reg_info*** to find the registers used by the instruction. These registers can then be read and written with ***SIM_instruction_read_input_reg*** and ***SIM_instruction_write_input_reg***.

Under special circumstances the instruction cannot be decoded until the value of a certain register that determines the instruction is known. For example on the SPARC architecture the value of the asi register can change the instruction from a normal memory operation into a block operation. In this case ***SIM_instruction_proceed*** and ***SIM_instruction_decode*** will return an error code until the value of the asi register is known.

4.3.3 Execution Phase

Instruction semantics are modeled during the execution phase. Thus input values are used to produce output values. The input values used are collected from previously executed instructions by reading from the pool of internal registers or from the architectural register state. The output values are written to the internal registers. As stated before Simics has no restrictions on the number of internal registers.

Before an instruction can execute all the input values must have been produced, i.e. all dependences must be fulfilled. This is done either by executing all instructions that the instruction depends on or by explicitly setting the value of an input register with ***SIM_instruction_write_input_reg***. As with the program counter the latter case will also make the instruction speculative until an earlier instruction produces the same value. This is how value speculation can be handled.

Instructions that access memory may stall if the connected memory hierarchy returns a stall time. ***SIM_instruction_proceed*** and ***SIM_instruction_execute*** will then return a status code to signal this.

As with the fetch and decode phases, a status code will be returned if the execution phase does not complete due to an exception.

4.3.4 Retire Phase

During the Retire phase, all stores that were speculatively executed and kept in the LSQ are sent to memory, one by one and in order if the instruction executed several.

Before an instruction can retire it must have become non-speculative. Retiring stores can be done out-of-order since the LSQ will take care of reordering stores that are conflicting. Note that starting the Retire phase makes the instruction unsquashable since the architectural state will be modified.

An instruction may stall during the Retire phase since memory operations are sent to the memory hierarchy. The status codes are the same as for the Execute phase. A retiring store may trigger an exception (like a memory parity error). In this case, it will be marked as faulting with status executed.

4.3.5 Commit Phase

Instruction output is written back to the architectural state during the commit phase. When committed, the instruction is automatically removed from the instruction tree, but the data structures remain until explicitly deallocated by calling ***SIM_instruction_end***. Instructions marked as speculative cannot be committed. However the user can force speculative commit by calling ***SIM_instruction_force_correct*** to remove the speculative status before committing. This is strongly discouraged since it may lead to incorrect execution. In addition, an instruction can only be committed if it is at the root of the instruction tree and it has only one child instruction. The one child limit is to avoid the creation of two separate instruction trees.

Committed instructions cannot be discarded since their output have been copied to architectural state and thus cannot be rolled back.

4.3.6 Exceptions

Exceptions may occur at any phase of instruction operation. Exceptions are signaled by a return code from ***SIM_instruction_proceed*** or its shortcuts. Once an exception has been signaled the instruction cannot proceeded to further phases. The exception should be handled by calling ***SIM_instruction_handle_exception*** on the faulting instruction. This call will set

the program counter to the first instruction in the corresponding exception handler. Currently this function requires that the instruction tree contains only the faulting instruction. All earlier instructions must have been committed and all later instructions need to be discarded. All previous instruction must be committed as Simics can only service exception handlers in order and non-speculatively. This cannot be guaranteed unless the instruction with the exception is the oldest instruction in the tree.

4.3.7 Interrupts

Interrupts differ from exceptions in that they are inherently asynchronous. Thus while the handling mechanism is similar the signaling mechanism is quite different. Simics signals that an interrupt has occurred by raising an Asynchronous_Trap hap. The user must install a handler for this hap. The suggested operation is that the hap handler set a flag visible to the cycle handler. Once the cycle handler is aware of an interrupt it may choose to service the interrupt by calling ***SIM_instruction_handle_interrupt*** at any time, subject to one restriction - the instruction tree must be empty. When ***SIM_instruction_handle_interrupt*** is called Simics will cache the current value for the next instruction and replace it with the address of the first instruction in the interrupt vector. When the interrupt completes the cached address will be used to return control to the interrupted location.

4.4 Discarding Instructions

When speculation is used, the CPU model may cause wrong path or otherwise incorrect execution and the effects of these instructions must be discarded. This is done in Simics by calling ***SIM_instruction_squash*** on an instruction. This function will remove the instruction and all its descendants from the instruction tree and deallocate the data structures. There is no need to call ***SIM_instruction_end***.

Memory operations that are associated with squashed instructions that have been issued from the consistency controller to the rest of the memory hierarchy will continue as normal, though no values will be written. Memory instruction that have not issued from the consistency controller will simply be removed. Thus from the perspective of the memory hierarchy, memory transactions are either executed completely or not at all.

4.5 Synchronous Instructions

Simics does not currently have the ability to rename all architectural state, thus there are a limited number of instructions that write to the architectural state immediately. These instructions are called synchronous instructions as they must be run synchronously. ***SIM_instruction_is_sync*** can be used to determine if an instruction is synchronous. These instructions need to be executed non-speculatively and when all earlier instructions in the tree have been executed or committed. This is to ensure that the immediate changes to the architectural state caused by the instruction are never incorrect. If a synchronous instruction is executed in an illegal way ***SIM_instruction_proceed*** will return an error code.

Note that this is a Simics limitation and not an architectural limitation. In appendix B and appendix C all synchronous instructions are listed for SPARC and x86 respectively.

4.6 Limitations in the Simics MAI Implementation

- Simics does not currently have an address calculation phase which means that the address for a store instruction is calculated when *all* input registers are known, not as soon as the registers used in the address calculation are known. This may cause a store to prevent other operations from running as soon as they could have, and thus limit the parallelism achievable by the model more than what is theoretically necessary.
- (*SPARC only*): Block loads block operation and stores obey register dependences, but the hardware does not. As long as the running code is written so that it does not rely on pipeline effects, this limitation should not be a problem.
- The instructions tree is not saved in a checkpoint. Only the committed architectural state and retired stores are saved.
- Some x86 transactions are non-stallable and will bypass the LSQ. This includes stack transactions performed when taking an interrupt or an exception, hardware table-walks, saving the entire fp state or doing a 10-byte fp read/write.

Chapter 5

The Memory System

This section will describe in details the elements that are used in Simics MAI memory system.

5.1 Load-Store Queue

The LSQ built into Simics is based on the instruction tree. Store transactions are kept in the tree to make it possible to inspect the store queue state in the different speculation paths.

Inserting a store transaction in the queue takes no time and the LSQ has an infinite size. It is up to the user module to set delays and restrictions to limit the LSQ.

The LSQ enforces program-order consistency. The following rules apply:

Loads from memory

A load transaction is allowed to execute only if all previous stores have been executed (i.e., there have been inserted in the LSQ). If there are instructions potentially performing stores higher up in the execution path that are still at the decode phase, the load is blocked until the blocking stores have been executed.

If the load is not blocked, it is matched against speculative stores in the LSQ (in the corresponding execution path). If matches are found, the data is retrieved from the LSQ. If more data is needed, the load is sent to memory. The resulting data is merged with the LSQ data and the result is returned to the processor.

Loads from devices

A load transaction that accesses a device is only allowed to execute as root of the tree (which makes it non-speculative).

Stores to memory

When the instruction enters the execute phase, stores are inserted in the LSQ.

When the instruction enters the retire phase, the corresponding stores are sent to memory (in order if the instruction has performed several). The LSQ searches for potential conflicting stores and forces stores that would overlap each other to execute in-order.

Stores to devices
A store transaction that accesses a device is only allowed to execute as root of the tree (which makes it non-speculative).

Atomic instructions
Atomic instructions will be blocked if they may conflict within a certain granularity with an earlier load in the tree that has not been executed or an earlier store in the tree that has not been retired. This is because atomic instructions lock a ram region when they have reached the execute phase (see section 5.2) which means that a deadlock situation may occur if an atomic instruction executes before an earlier conflicting instruction. The earlier instruction will in this case wait for the lock to be released and the atomic instruction will wait for the earlier instruction to complete according to the above rules of the LSQ. To avoid this atomic instructions will have to wait until there are no conflicts. Due to the locking mechanism the load part of an atomic instruction will always be sent to memory although all its data can be found in the LSQ.

The granularity can be set by an attribute in the processor object, *<processor>.lock_granularity*. If set to zero atomic instructions will not be blocked and it is up to the processor model to avoid deadlocks. If set to non-zero it should be set to the same granularity as set for the ram objects. The default granularity is 8 bytes which is the same as the default granularity for the ram object.

Some memory transactions bypass the LSQ:

- Instruction fetches are always sent directly to memory. If you want your model to handle self-modifying code (instead of letting the software synchronize this itself), you will need to reorder instruction fetches when they arrive to the memory hierarchy.
- Control (cache lock, flush, ...) and prefetch transactions bypass the LSQ as well.
- Some special x86 transactions may bypass the LSQ but there are usually associated to synchronizing instruction so it will not matter. The model limitations are listed in section 4.6.

5.1.1 Disabling the Load-Store Queue

The internal LSQ can be disabled by setting the attribute *lsq_enabled* to 0 in each CPU object. When the LSQ is disabled all memory transactions are sent to memory during the execution phase. The retire phase thus becomes superfluous and proceeding instructions through it will have no effect.

Note that when the LSQ is turned off program order consistency will **not** be maintained any longer by Simics. It is up to the timing model to ensure that memory operations does not complete in wrong order. It is therefore strongly recommended that the LSQ never disabled unless you are 100% sure of what you are doing.

The main reason for disabling the LSQ would be if a model of a memory system should be simulated that is more relaxed than what is allowed by the internal LSQ.

5.2 Atomic Instructions

There is a memory locking mechanism implemented for atomic instructions that prevents processors in a multi-processor system from breaking atomic properties. An atomic instruction typically consists of a load and a store part that must be performed without any intervening conflicting transactions. MAI handles this by locking a ram region at a configurable granularity when the first access is issued. The lock is released as soon as the second access is finished or if the instruction is squashed. Any conflicting transaction to the same memory region issued between the two by any processor will stall until the lock is released. The granularity can be set in each ram object to control the size of a region. If the size is set to 0 no locking will be performed and it is then up to the timing model to handle the locking. The default granularity is set to 8 bytes.

5.3 Consistency Controller

Note: In previous versions of Simics, the consistency controller was responsible for keeping program order consistency. This is now performed by the LSQ so no consistency controller is needed to run one processor systems.

When simulating multi-processors systems with Simics MAI, you may want to apply a stronger consistency model then the one the LSQ is providing. The consistency controller module provides some default models.

The consistency controller can be controlled through attributes (setting the an attribute to 0 will imply no constraint):

load-load
if set to non-zero loads are issued in program order

load-store
if set to non-zero program order is maintained for stores following loads

store-load
if set to non-zero program order is maintained for loads following stores

store-store
if set to non-zero stores are issued in program order

prefetch
prefetching (see explanations below)

Instructions that issues both loads and stores, e.g. swap instructions, will have both the load and the store status. Thus setting any of the first four attributes will order all such instructions.

The *prefetch* attribute is orthogonal to all other consistency controller attributes. When not enabled the consistency controller does not issue any memory transaction to the remainder of the memory hierarchy until they are guaranteed to obey the consistency rules. When

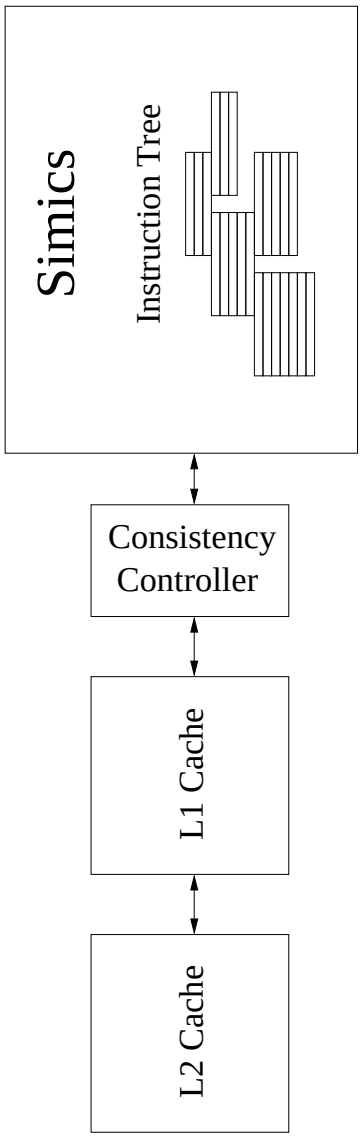

Figure 5.1: Consistency Controller

enabled the prefetch option causes a prefetch to be issued to the rest of the memory hierarchy for all operations where the target address is ready before the consistency controller has guaranteed correctness. As prefetches are nonbinding, it is always correct for any portion of the memory system to ignore them. Note: prefetches are not allowed to stall and they are only sent once for each instruction.

The source code for the consistency controller is available so it is easy to change its behavior in detail. See `simics/src/extensions/consistency-controller`.

5.4 Generic Cache OOO

A special cache model called **g-cache-ooo** is provided with Simics MAI to handle multiple outstanding transactions and out-of-order execution. It is very similar to **g-cache** described in *Simics User Guide* so we describe here only the differences that are introduced by the special MAI requirements.

Although **g-cache** and **g-cache-ooo** have a very similar interface (in terms of attributes and commands), they do not handle transactions in the same way. Whereas **g-cache** takes care of a transactions in one pass, computing all the penalties and performing all the state changes as the transaction goes through its different phases, **g-cache-ooo** tries to present a more credible model for out-of-order execution. Each phase of a transaction in the cache is considered and handled separately. A transaction may trigger a copy-back that will last some cycles, then a fetch will be sent to the next level stalling the cache for some more cycles, and finally the result will be returned. During these phases, **g-cache-ooo** ensures that other transactions behave in a sensible manner: for example, if two transactions want to fetch the same cache line, the second transaction is considered as a delayed hit and its value will be returned when the first (missing) transaction is completed; cache lines are locked as long as the transaction using them is on-going.

g-cache-ooo supports the same options as **g-cache** except:

- MESI is not implemented.
- There is no *penalty_read_next* and *penalty_write_next* attributes. You should connect a next level memory hierarchy if you want transactions to the next level to stall longer. The **trans-staller** module can be used to set a simple delay.
- **g-cache-ooo** can not use the STCs to get improve the speed of the simulation.

Some new, MAI specific features have been introduced:

- The attributes *config_read_per_cycle* and *config_write_per_cycle* limit the number of read and write transactions that can be initiated in the cache for a given cycle.
- The attributes *config_max_out_trans* limits the number of outstanding transactions that can be present in the cache.

g-cache-ooo can be used with any Simics configuration, even without MAI. However, since it relies quite heavily on the ability to stall transactions to handle state changes, only stallable transactions are considered cacheable. If you want to warm-up your caches with

a normal run and wish to include transactions that are normally not stallable (and thus not cacheable) like x86 instruction fetches, you should:

1. Set all penalties in your memory hierarchy to 0.
2. Set the attribute *config_accept_no_stall* to 1 to tell the cache to accept non-stallable transactions.

5.5 Instruction Fetches

Instruction fetches are not sent to the memory hierarchy by default. To enable this set the attribute *instruction-fetch-mode* to the string `"instruction-fetch-trace"` in each processor object or use the command **instruction-fetch-mode** to enable it on all processors, e.g.

```
instruction-fetch-mode instruction-fetch-trace
```

Note: The *instructions-fetch-mode* is a session attribute and Simics does not save session attributes in a checkpoint. Therefor it is necessary to reset this attribute or rerun the command each time a checkpoint is loaded. You can also add the attribute to each CPU object in the the checkpoint file by hand.

For instructions fetches that cross page boundaries (x86 only) the memory transaction will be split in two parts, where the first part will have *page_cross* set to one and the second part will have *page_cross* set to 2. For non-crossing transactions *page_cross* will be zero. This applies to data accesses as well.

Chapter 6

Getting Started with SPARC-V9 MAI

There are a number of pre-configured setups included in the Simics distribution that uses the MAI to simulate out-of-order processors. They are all examples on how to use the API in different ways. The scripts reside in the same directory as the ordinary Simics scrips for a specific target. The name of the scripts contains `ooo` (for *out-of-order*). Here follows a short description of each script and how they can be further configured:

- `simics/targets/sunfire/bagle-ooo-common.simics`
- `simics/targets/sunfire/donut-ooo-common.simics`
- `simics/targets/serengeti/sarek-ooo-common.simics`

  These scripts uses the module **ooo-micro-arch** (see the source code in `simics/src/extensions/ooo-micro-arch`) to demonstrate how MAI works. The model can fetch, execute, and commit a configurable number of instructions per cycle. No branch speculation is performed, thus if an unresolved branch is found the fetches are stalled until the outcome of the branch is determined.

  If an exception occurs the instruction tree is drained and all the speculative instructions beyond the one that caused the exception are discarded.

  A pipeline is modeled with a combined fetch/decode stage, an execute state, and a commit stage.

  Each processor gets an object of the **ooo_micro_arch** class attached to it that handles the simulation. These objects have some attributes that can be changed to alter the behavior of the model:

  - *<ooo_micro_arch>.fetches_per_cycles* controls the number of instructions that can be fetched and decoded per cycle. Default is 1.
  - *<ooo_micro_arch>.execute_per_cycles* controls the number of instructions that can be executed per cycle. The instructions can be dependent of each other. Default is 1.
  - *<ooo_micro_arch>.commits_per_cycles* controls how many instructions that can be committed per cycle. Default is 1.

- `simics/targets/sunfire/bagle-ma-common.simics`
- `simics/targets/sunfire/donut-ma-common.simics`
- `simics/targets/serengeti/sarek-ma-common.simics`

  These scripts use the **sample-micro-arch** module (see `simics/src/extensions/sample-micro-arch`). The processors modeled can fetch/decode, execute, and commit a configurable number of instructions per cycle.

  The model has a simple branch-predictor that uses a hash table (Branch Target Buffer) to lookup the target address from the address of the branch instruction. This allows for branch speculation. The hash table is updated for every successfully committed branch.

  Besides speculating on the target address, the model also speculate fall through for every branch. This way two possible execution paths are created for every branch. This makes the instruction tree into a binary tree. The number of instructions executed and fetch per cycle is actually per branch in the instruction tree.

  There is a compile time switch available called `VALUE_PREDICTION` that can be defined to switch on value prediction of loads. It works like a small cache that maps logical addresses to values. When a load is issued, the cache is looked up first to quickly get value that can be used by later instructions. When the load is finished the speculated value is checked against the real value. If they mismatch, the later instructions need to be squashed.

  Each processor gets an object of the **sample_micro_arch** class attached to it that handles the simulation. The class implements the following attributes:

  - *<sample_micro_arch>.fetches_per_cycles* controls the number of instructions that can be fetched and decoded per cycle. Default is 4.
  - *<sample_micro_arch>.execute_per_cycles* controls the number of instructions that can be executed per cycle. The instructions can be dependent of each other. Default is 4.
  - *<sample_micro_arch>.retires_per_cycles* controls how many stores that can be retired to memory per cycle. Default is 4.
  - *<sample_micro_arch>.commits_per_cycles* controls how many instructions that can be committed per cycle. Default is 4.
  - *<sample_micro_arch>.out_of_order_retire*. If non-zero the retire phase can be performed out of order. Default is 0.

The following attributes in each CPU object can also be used to further configure the models:

- *<processor>.reorder_buffer_size* controls the total number of instructions that fit in the instruction tree.

- *<processor>.auto_speculate_cwp*. If set to non-zero the CWP register is automatically speculated, i.e. if a save instruction is encountered in the instruction stream the value of the CWP register will automatically be incremented (modulo the number of windows) for the following instructions, and if a restore or a return instruction is found CWP will be decremented automatically.

Chapter 7

Getting Started with x86 MAI

The *Enterprise* in `[simics]/targets/x86-440bx/` comes pre-configured with a simple MAI model: `enterprise-ma-common.simics`. You can start it as you would start any Simics script:

```
# ./simics -ma targets/x86-440bx/enterprise-ma-common.simics
Checking out a license... done: academic license.

  +-----------------+    Copyright 1998-2005 by Virtutech, All Rights Reserved
  |   Virtutech     |    Version: Simics 3.0.0
  |     Simics      |    Build: trunk Tue Sep 13 10:03:44 2005 UTC
  +-----------------+
    www.simics.com       "Virtutech" and "Simics" are trademarks of Virtutech AB

Type 'copyright' for details on copyright.
Type 'license' for details on warranty, copying, etc.
Type 'readme' for further information about this version.
Type 'help help' for info on the on-line documentation.

simics>
```

If the log level of the sample x86 model is set to 2, it will print out how many steps it has executed every 65536 cycles. Since the speed of the model depends a lot on the executed instructions, it is a useful indication of how far the simulation has run:

```
simics> ma_cpu0.log-level 2
[ma_cpu0] Changing log level: 1 -> 2
simics> c
[ma_cpu0 info] cycle: 65535, steps: 128584
[ma_cpu0 info] cycle: 131071, steps: 325192
[ma_cpu0 info] cycle: 196607, steps: 521800
```

Note: The P4 memory hierarchy example and the scripts to handle the change from in-order execution to MAI in Simics 2.x have not been ported yet to the new `targets` scripts, which is why they are not presented here.

Chapter 8

Using Simics with MAI

Running an MAI model in Simics affects Simics's user interface. Whereas in in-order Simics changes are atomic and always visible to the user, in Simics MAI multiple instructions can be in-flight at the same time and several instructions can be committed in the same cycle. The consequence is that standard Simics commands for inspecting the machine state will now only inspect the *committed state* of the CPU. There are several other small differences when running MAI compared to in-order Simics. These differences are described in the following sections.

8.1 Starting Simics

When using MAI Simics must be started with the *-ma* flag, otherwise the standard in-order version will be used. Make sure you start Simics with a start script that is intended for MAI. All such scripts provided by Virtutech contains `-ma-` or `-ooo-`. For example,type:

```
# ./simics -ma targets/serengeti/sarek-ma-common.simics
```

This will start Simics in MAI mode. The script will add a cache hierarchy with one **g-cache-ooo**. A **sample-micro-arch** object will also be added that performs the micro-architectural simulation. Its cycle handler will be posted on cpu0.

8.2 Inspecting State

A command called **print-instruction-queue** is provided to print out the current execution tree, displaying input and output register values according to the phase of execution of the instructions in flight. Continue using the started Simics session from section 8.1 and run for a while, then inspect the instruction tree by issuing the following commands:

```
simics> run-cycles 2000
simics> print-instruction-queue
```

This will print all the inserted instructions in the tree. Branches will be indented if they exists. Using the **-v** flag to **print-instruction-queue** will tell the command to print the values of input and output registers and the contents of the LSQ as well.

print-instruction-queue can also be invoked for a special processor, e.g.

```
simics> cpu0.print-instruction-queue
```

8.3 Running and Stepping

The **run-cycles** command is used in the example above. It will run a particular number of cycles before returning to prompt. It is more appropriate to use this command than the **run** (**continue**) command when running MAI since the **run** command will run a certain number of instructions (steps) and this may not add up to an even number of cycles causing a cycle to be incomplete. The same holds for **step-instruction** and step breakpoints inserted by the **step-break** and **step-break-absolute** commands.

Steps are consumed just before an instruction is committed, or an exception or interrupt is handled. Breaking after a number of steps may force the MAI module to break inside a cycle. It is up to the module to decide what action to take if this happens. ***SIM_instruction_proceed***, ***SIM_instruction_commit***, ***SIM_instruction_handle_exception***, and ***SIM_instruction_handle_interrupt*** will return special value if the simulation should be stopped at the next step count. The module can call ***SIM_break_cycle(cpu, 0)*** or ***SIM_break_simulation*** and then exit the cycle handler to return to prompt. The **sample-micro-arch** and **ooo-micro-arch** do this after printing a warning message.

To advance the simulation a single cycle use the **step-cycle** command. A neat combination of commands is:

```
simics> sc; piq -v
```

sc is short for **step-cycle** and **piq** is short for **print-instruction-queue**. With this it is easy to monitor changes in the instruction tree for each new cycle.

For multiprocessor systems there is a special command called **step-cycle-single** or **scs** for short that steps the first processor one cycle, then if reinvoked it steps the next processor one cycle etc. The current fronted processor is changed between each invocation making inspection commands like **print-instruction-queue** to work on the newly stepped processor:

```
simics> scs; piq -v
```

8.4 Breakpoints

Code breakpoints and magic breakpoints will trigger just before the instruction is committed or an exception is handled. ***SIM_instruction_proceed***, ***SIM_instruction_commit***, and

SIM_instruction_handle_exception will signal this by returning a special value and it is up to the processor module to exit the cycle.

Time breakpoints will work fine and the preferred way of breaking Simics in MAI. Step breakpoints may break inside a cycle as described in the previous section.

Pressing control-C will always break at an even cycle boundary.

8.5 Checkpointing

The recommended way of working with MAI is to first position the workload by running standard in-order Simics. Using in-order simulation is magnitudes faster than running in micro-architectural mode.

We recommend the following steps for your simulation setup:

1. Boot the machine in in-order Simics. Use one of the provided scripts in the target directory. Most of the machines will log in automatically.

2. Import your workload by using SimicsFS for example. Take a checkpoint.

3. Add caches to the configuration using a script. Warm the caches by running up to the point where your actual workload starts. If you are using other components that needs warming this may be done in this phase too. I branch predictor could for example be warmed up by using the tracing facilities in Simics. This time run with the *-stall* option. Cache simulation is not supported by the normal execution mode. Take another checkpoint.

4. Start Simics in MAI mode from the second checkpoint:

   ```
   # ./simics -ma -c checkpoint2
   ```

   Use a script to add your micro-architectural model and possibly consistency controllers to the configuration. Now you can start simulating.

Appendix A

Micro Architectural Interface API

Here follows a description on all function in the Micro Architecture Interface. Many of the API functions use the enum type instruction_error_t for signaling error codes. The type is defined like this:

```
typedef enum instruction_error {
        Sim_IE_OK = 0,
        Sim_IE_Unresolved_Dependencies,
        Sim_IE_Speculative,
        Sim_IE_Stalling,
        Sim_IE_Not_Inserted,      /* trying to execute or squash an
                                     instruction that is inserted. */
        Sim_IE_Exception,         /* (SPARC-V9 only) */
        Sim_IE_Fault = Sim_IE_Exception,
        Sim_IE_Trap,              /* (X86 only) Returned if a trap is
                                     encountered */
        Sim_IE_Interrupt,         /* (X86 only) Returned if an interrupt is
                                     waiting and interrupts are enabled */

        Sim_IE_Sync_Instruction,  /* Returned if sync instruction is
                                     not allowd to execute */
        Sim_IE_No_Exception,      /* Returned by SIM_instruction_
                                     handle_exception */
        Sim_IE_Illegal_Interrupt_Point,
        Sim_IE_Illegal_Exception_Point,
        Sim_IE_Illegal_Address,
        Sim_IE_Illegal_Phase,
        Sim_IE_Interrupts_Disabled,
        Sim_IE_Illegal_Id,
        Sim_IE_Instruction_Tree_Full,
        Sim_IE_Null_Pointer,
        Sim_IE_Illegal_Reg,
        Sim_IE_Invalid,
        Sim_IE_Out_of_Order_Commit,
        Sim_IE_Retired_Instruction, /* try to squash a retiring instruction */
        Sim_IE_Not_Committed,       /* Returned by SIM_instruction_end */
        Sim_IE_Code_Breakpoint,
        Sim_IE_Mem_Breakpoint,
        Sim_IE_Step_Breakpoint,
        Sim_IE_Hap_Breakpoint
} instruction_error_t;
```

A.1 Micro Architecture API Functions

SIM_instruction_begin()

NAME

SIM_instruction_begin — begin an instruction

SYNOPSIS

```
instruction_id_t
SIM_instruction_begin(conf_object_t *cpu);
```

DESCRIPTION

In the Micro Architectural Interface instructions passing through the processor pipeline are identified by an instruction id. This function creates a new id and allocates the necessary data structures to handle the instruction. All the API functions referring to the instruction will take the instruction id as a parameter.

To assign the id to a particular instruction in memory, you should speculate on the program counter input register with ***SIM_instruction_write_input_reg***. Instructions should also be linked together to form execution paths, see ***SIM_instruction_instert*** for details.

Instructions are deallocated by ***SIM_instruction_end***.

EXCEPTIONS

General Thrown if the pre-allocated space for the instruction queue which is set by the processor attribute *reorder_buffer_size* is full.

RETURN VALUE

A new instruction id.

SEE ALSO

SIM_instruction_insert, SIM_instruction_proceed, SIM_instruction_end

SIM_instruction_child()

NAME

SIM_instruction_child, SIM_instruction_parent — get the parent or child

SYNOPSIS

```
instruction_id_t
SIM_instruction_child(instruction_id_t ii, int n);
```

```
instruction_id_t
SIM_instruction_parent(instruction_id_t ii);
```

DESCRIPTION

SIM_instruction_child returns the child *n* (counting started from 0) for instruction *ii*. Child 0 will always be valid if the instruction has a child, i.e. if child 0 is squashed child 1 will become 0 etc.

SIM_instruction_parent returns the parent instruction.

EXCEPTIONS

Index Thrown if ii is illegal.

RETURN VALUE

The instruction id of the requested instruction, or 0 if there is none.

SIM_instruction_cpu()

NAME

SIM_instruction_cpu — get the CPU of an instruction

SYNOPSIS

```
conf_object_t *
SIM_instruction_cpu(instruction_id_t ii);
```

DESCRIPTION

Returns the CPU that the instructions *ii* was created on using ***SIM_instruction_begin***.

EXCEPTIONS

Index Thrown if ii is illegal.

RETURN VALUE

The CPU of the instruction ii.

SIM_instruction_end()

NAME

SIM_instruction_end — deallocate an instruction

SYNOPSIS

```
instruction_error_t
SIM_instruction_end(instruction_id_t ii);
```

DESCRIPTION

This function deallocates the data structures previously allocated by ***SIM_instruction_begin***. The instruction can be ended if it is not yet inserted in the instruction tree or if it is committed (***SIM_instruction_proceed***, ***SIM_instruction_commit***). To remove an inserted instruction use ***SIM_instruction_squash***.

ii is the id of the instruction to deallocate.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

`Sim_IE_OK` on success,
`Sim_IE_Not_Committed` if the instruction is not committed.

SEE ALSO

SIM_instruction_begin, SIM_instruction_proceed, SIM_instruction_squash

SIM_instruction_force_correct()

NAME

SIM_instruction_force_correct — remove speculative status

SYNOPSIS

```
void
SIM_instruction_force_correct(instruction_id_t ii);
```

DESCRIPTION

Note: this function was previously called ***SIM_instruction_set_right_path***.

This function removes the speculative status of the instruction. This can be used to model an architecture which is not consistent with program order execution. Use this function with care since it may lead to incorrect results.

ii is the id of the instruction to check,

EXCEPTIONS

Index Thrown if *ii* is illegal.

SIM_instruction_get_field_value()

NAME

SIM_instruction_get_field_value — read value of format field

SYNOPSIS

```
integer_t
SIM_instruction_get_field_value(instruction_id_t ii,
                                const char *field_name);
```

DESCRIPTION

This function is deprecated. Use ***SIM_instruction_opcode*** instead.

Reads the value of a format field. Fields that represent signed constants will automatically be sign extended. *ii* is the instruction to on CPU *cpu* to extract the field from.

Supported fields for the SPARC v9 target are (see the SPARC Architecture Manual, Version 9): `a`, `cc0`, `cc1`, `cc2`, `cmask`, `cond`, `d16hi`, `d16lo`, `disp19`, `disp22`, `disp30`, `fcn`, `i`, `imm22`, `imm_asi`, `mmask`, `op`, `op2`, `op3`, `opf`, `opf_low`, `p`, `rcond`, `rd`, `rs1`, `rs2`, `shcnt32`, `shcnt64`, `simm10`, `simm11`, `simm13`, `sw_trap`, `x`.

The only supported field for the x86 target is `prefix` which is a bit field that can be tested with the following constants: `SIM_DI_PREFIX_F0_BIT`, `SIM_DI_PREFIX_F2_BIT`, `SIM_DI_PREFIX_F3_BIT`, `SIM_DI_PREFIX_CS_BIT`, `SIM_DI_PREFIX_SS_BIT`, `SIM_DI_PREFIX_DS_BIT`, `SIM_DI_PREFIX_ES_BIT`, `SIM_DI_PREFIX_FS_BIT`, `SIM_DI_PREFIX_GS_BIT`, `SIM_DI_PREFIX_OPERAND_SIZE_BIT`, `SIM_DI_PREFIX_ADDRESS_SIZE_BIT` and `SIM_DI_PREFIX_SSE_BIT`.

Note that certain fields are only available for certain instructions, thus accessing `simm13` when `i` is 0 will be illegal.

The instruction has to be decoded in order to use this function. See ***SIM_instruction_proceed***.

EXCEPTIONS

General Thrown if the instruction is not decoded.
Index Thrown if *ii* is illegal, or if the field is illegal.

RETURN VALUE

The value of the field.

DISCLAIMER

This function may be changed in future and be replaced by an alternative mechanism for reading the fields.

SIM_instruction_get_reg_info()

NAME

SIM_instruction_get_reg_info — get info about instruction register

SYNOPSIS

```
reg_info_t *
SIM_instruction_get_reg_info(instruction_id_t ii,
                             int n);
```

DESCRIPTION

This function can by used to get information about which register an instruction uses as input and/or output. The function takes as arguments the instruction id, *ii*, and a number for the *n*:th used register. If successful, a pointer to a `reg_info_t` will be returned. The structure is only available for immediate use. 0 is return if there is no n:th register. Typically this function is used in a loop to retrieve all registers until it returns 0.

The `reg_info_t` is defined like this:

```
struct reg_info {
        register_type_t type;    /* register type */
        register_id_t id;        /* register id */
        unsigned input:1;        /* used as input */
        unsigned output:1;       /* used as output */
};
```

The `register_id_t` refers to the different registers of the architecture in question. Only registers that are targets for register renaming are reported by this function.

EXCEPTIONS

General If the instruction is not decoded.
Index Thrown if *ii* is illegal.

RETURN VALUE

pointer to a `reg_info_t` structure or NULL if there is no n:th register.

SEE ALSO

SIM_instruction_read_input_reg

SIM_instruction_get_user_data()

NAME

SIM_instruction_get_user_data, SIM_instruction_set_user_data — get/set instruction user data

SYNOPSIS

```
lang_void *
SIM_instruction_get_user_data(instruction_id_t ii);


void
SIM_instruction_set_user_data(instruction_id_t ii,
                              lang_void *data);
```

DESCRIPTION

These two functions are used to set and get user defined data for instruction identified by *ii*. This is a convenient way to add extra information to an instruction. Note that this information needs to be deallocated by the user before ***SIM_instruction_end*** is called.

If an instruction is squashed the Instruction_Squashed hap should be used to handle the deallocation of user data.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

SIM_instruction_get_user_data returns the user data.

SIM_instruction_handle_exception()

NAME

SIM_instruction_handle_exception — handle exception

SYNOPSIS

```
instruction_error_t
SIM_instruction_handle_exception(instruction_id_t ii);
```

DESCRIPTION

When SIM_instruction_proceed or one of its shortcuts returns `Sim_IE_Exception` an exception has occurred. The exception should be handled by an explicit call to ***SIM_instruction_handle_exception*** with the faulting instruction's id as input.

The CPU needs to be synchronized to handle an exception. This is achieved by draining the instruction tree so that all instructions before the faulting one are committed and all later instructions are discarded. Exceptions can also be handled if the instruction is not inserted in the tree and the tree is empty. This can happens if an instruction gets an exception in the fetch or decode phase and the instruction is not inserted in the tree.

Speculative instructions that cause exceptions cannot be handled. If this is what the user wants ***SIM_instruction_force_correct*** can be used to remove the speculative status before the exception is handled. This is strongly discouraged since it may lead to incorrect execution.

The effect of calling ***SIM_instruction_handle_exception*** is to set the program counter to the first instruction in the corresponding exception handler and to update other architectural state that are affected when an exception occurs.

The exception is automatically "committed" so there is no need to call ***SIM_instruction_proceed*** or ***SIM_instruction_commit***.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

`Sim_IE_OK` on success,
`Sim_IE_Code_Breakpoint` if the instruction had a code breakpoint,
`Sim_IE_Step_Breakpoint` if a step breakpoint was encountered,
`Sim_IE_Hap_Breakpoint` if the exception caused an hap breakpoint,
`Sim_IE_No_Exception` if trying to handle an the exception on a non-faulting instruction,
`Sim_IE_Speculative` if trying to handle an exception from a speculative instruction,
`Sim_IE_Illegal_Exception_Point` if the tree contains other instructions.

SIM_instruction_handle_interrupt()

NAME

SIM_instruction_handle_interrupt — handle interrupt

SYNOPSIS

```
instruction_error_t
SIM_instruction_handle_interrupt(conf_object_t *cpu,
                                 exception_type_t vector);
```

DESCRIPTION

Interrupts are caught in the Micro Architectural Interface by installing an callback handler on the Core_Asynchronous_Trap hap. This callback will be called when an interrupt or an asynchronous trap occurs. The callback gets the interrupted CPU and the trap vector number associated with the interrupt as arguments (see the hap documentation for details).

To handle the interrupt ***SIM_instruction_handle_interrupt*** should be called with the CPU and the trap vector as arguments. The effect of this call is to set the program counter to the first instruction in the corresponding trap handler and to update other architectural state that are affected when an interrupt occurs.

Currently the instruction tree needs to be empty when an interrupt is handled. This is achieved either by squashing all instructions or letting all of them commit before ***SIM_instruction_handle_interrupt*** is called.

Interrupts may be disabled (by software or hardware) after the hap triggers but before ***SIM_instruction_handle_interrupt*** is called, if this is the case `Sim_IE_Interrupts_Disabled` will be returned. The interrupts should then be handled as soon as the interrupts are enabled again.

RETURN VALUE

`Sim_IE_OK` on success,
`Sim_IE_No_Exception` if vector is `No_Exception`,
`Sim_IE_Interrupts_Disabled` if interrupts are disabled,
`Sim_IE_Illegal_Interrupt_Point` if trying to handle an interrupt when the instruction queue is not empty.

SIM_instruction_id_from_mem_op_id()

NAME

SIM_instruction_id_from_mem_op_id — return instruction id

SYNOPSIS

```
instruction_id_t
SIM_instruction_id_from_mem_op_id(conf_object_t *cpu, int mem_trans_id);
```

DESCRIPTION

Given the memory_transaction id *mem_trans_id* this function returns the corresponding instruction id.

RETURN VALUE

The instruction id of the instruction corresponding to the memory transaction id, or NULL if no instruction match.

SIM_instruction_insert()

NAME

SIM_instruction_insert — link instructions

SYNOPSIS

```
void
SIM_instruction_insert(instruction_id_t parent_ii,
                       instruction_id_t ii);
```

DESCRIPTION

This function links together instructions created by ***SIM_instruction_begin***. The linking helps the user to keep track of dependences between instructions and as default, Simics will not accept any dependence violation. To model different (speculative) execution paths instructions can be linked together to form a tree (instruction tree). The instructions in a tree branch should be ordered in program execution order.

ii is the instruction to insert and *ii_parent* is the instruction id of the parent instruction. If zero the instruction will be inserted as child to the last inserted one, or as the root if the the tree is empty.

If the parent instruction already has a child the new instruction will be added as another child, thus creating an alternative execution path for the parent.

Instructions can be inserted in the tree after they are fetched and decoded (using ***SIM_instruction_proceed*** or similar), but must be inserted before they are executed. This is useful if different instruction should be inserted at different places. For example the instruction after a delay slot have a different parent depending on whether the delay slot can be annulled or not.

Instructions are automatically removed from the tree when they are committed (***SIM_instruction_proceed***/***SIM_instruction_commit***) or squashed (***SIM_instruction_squash***).

SEE ALSO

SIM_instruction_begin, SIM_instruction_proceed

SIM_instruction_is_sync()

NAME

SIM_instruction_is_sync — check for sync instruction

SYNOPSIS

```
int
SIM_instruction_is_sync(instruction_id_t ii);
```

DESCRIPTION

Some instructions need to synchronize the pipeline and cannot run out of order. This function checks if the instruction identified by *ii* needs to synchronize the CPU. Such instructions need to be executed non-speculatively. The reason for this limitation is that synchronizing instructions may update CPU state that cannot be restored.

There are currently two types of synchronizing instructions, category 1 and category 2. An instruction in category 1 can only be executed if all previous instructions in the tree has been executed (reached the `Sim_Phase_Executed`) non-speculatively. An instruction in category 2 has to be alone in the instruction tree to be able to execute.

Note that this is a Simics limitation and not an architectural limitation. The instructions that can execute out of order may change in future.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

1 for category 1, 2 for category 2, and 0 for non-synchronous.

SIM_instruction_length()

NAME

SIM_instruction_length — return the instruction length

SYNOPSIS

```
int
SIM_instruction_length(instruction_id_t ii);
```

DESCRIPTION

Returns the instruction length of the instruction identified by *ii* (in bytes).

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The length of the instruction in the instruction queue.

SIM_instruction_nth_id()

NAME

SIM_instruction_nth_id — get id of nth instruction

SYNOPSIS

```
instruction_id_t
SIM_instruction_nth_id(conf_object_t *cpu, int n);
```

DESCRIPTION

Get the instruction id (`instruction_id_t`) of the instruction at position *n* in the "first" branch of the instruction tree of processor *cpu*. The root instruction has position 0. The first branch is defined to be the branch of the first child (number 0) in every instruction. See ***SIM_instruction_child***. If successful the function will return an non-zero id. For illegal positions zero is returned.

This function may only be useful if the tree is actually a list or if the root of the tree is requested.

RETURN VALUE

The id of the requested instruction or zero if non-existing.

SEE ALSO

SIM_instruction_id_from_mem_op_id

SIM_instruction_opcode()

NAME

SIM_instruction_opcode — get the opcode of the instruction

SYNOPSIS

```
attr_value_t
SIM_instruction_opcode(instruction_id_t ii);
```

DESCRIPTION

Returns the opcode of an instruction as a `attr_value_t`. If the instruction fits in a 64 bits integer, it will be returned as a `Sim_Val_Integer`, otherwise as a `Sim_Val_Data`.

If the opcode is unknown, i.e. the instruction has not yet been fetched, Sim_Val_Nil is returned.

RETURN VALUE

The value of the opcode.

SIM_instruction_phase()

NAME

SIM_instruction_phase — get instruction phase

SYNOPSIS

```
instruction_phase_t
SIM_instruction_phase(instruction_id_t ii);
```

DESCRIPTION

Returns the phase the instruction with id *ii* has successfully passed. See ***SIM_instruction_set_stop_phase*** for a description of the different phases.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The last completed phase.

SIM_instruction_proceed()

NAME

SIM_instruction_proceed, SIM_instruction_fetch, SIM_instruction_decode, SIM_instruction_execute, SIM_instruction_retire, SIM_instruction_commit — proceed to next instruction phase

SYNOPSIS

```
instruction_error_t
SIM_instruction_proceed(instruction_id_t ii);

instruction_error_t
SIM_instruction_fetch(instruction_id_t ii);

instruction_error_t
SIM_instruction_decode(instruction_id_t ii);

instruction_error_t
SIM_instruction_execute(instruction_id_t ii);

instruction_error_t
SIM_instruction_retire(instruction_id_t ii);

instruction_error_t
SIM_instruction_commit(instruction_id_t ii);
```

DESCRIPTION

This function advances the instruction *ii* to the next stop phase as set by ***SIM_instruction_set_stop_phase***.

The defined phases are the following:

```
typedef enum instruction_phase {
        Sim_Phase_Initiated,
        Sim_Phase_Fetched,
        Sim_Phase_Decoded,
        Sim_Phase_Executed,
        Sim_Phase_Retired,
        Sim_Phase_Committed,
        Sim_Phases
} instruction_phase_t;
```

For convenience there are some predefined functions that can be used to proceed to the phases above.

SIM_instruction_fetch(id) will proceed to Sim_Phase_Fetch.

SIM_instruction_decode(id) will proceed to Sim_Phase_Decoded.

SIM_instruction_execute(id) will proceed to Sim_Phase_Executed.

SIM_instruction_retire(id) will proceed to Sim_Phase_Retired.

SIM_instruction_commit(id) will proceed to Sim_Phase_Committed.

Fetching an instruction means that the memory used by the instruction is loaded. This may stall the CPU and ***SIM_instruction_proceed*** will signal this by returning `Sim_IE_Stalling`. The instruction can then be proceeded to the next phase as soon as the stalling period is over (see ***SIM_instruction_remaining_stall_time***).

To fetch an instruction a valid program counter is needed. The value of the program counter is read from the register bank for the first instruction in the tree and for instructions not inserted in the tree. For instructions that have an executed parent the program counter will have the value produced by the parent. ***SIM_instruction_write_input_reg*** can also be used to explicitly set the program counter. This is the only way to fetch instructions whose parent has not yet executed and thus not produced a new program counter. This will also mark the instruction as speculative. As soon as the parent instruction is executed the child will become non-speculative again if the produced value matches the one set (if the parent was non-speculative). For miss-matches we will get a speculative instructions that needs to be squashed.

All the above also apply to the next program counter if the architecture has delay slots.

When an instruction is fetched ***SIM_instruction_opcode*** can be used to retrieve the opcode.

Decoding means interpreting the fetched data and translate it to a determined instruction. It is then possible to use ***SIM_instruction_type*** to determine the type of the instruction and ***SIM_instruction_get_reg_info*** to find the registers used by the instruction.

Executing an instruction means that the actual operation is performed. Thus input values are used to produce output values. The input values used are collected from previously executed instructions (or the register bank) and the output values will be available for later instructions. Simics has no restrictions on the number of temporal values that can exists between instruction. This can be viewed as an unlimited resource of internal registers for register renaming.

To be able to execute the instruction all the input values must have been produced, i.e. all dependences must be fulfilled. This is done either by executing all instructions that the instruction depends on or explicitly set the value of an input register by using ***SIM_instruction_write_input_reg***. As with the program counter above the latter case will also make the instruction speculative until an earlier instruction produces the same value. This is how value speculation can be handled.

`Sim_IE_Unresolved_Dependences` will be returned if an instruction is not ready for execution.

Instructions that access memory may stall and `Sim_EI_Stalling` will be returned. As with instruction fetches the instruction can be proceeded again as soon as the stalling period is over.

Retiring an instruction sends all speculative stores to memory. The instruction must be non-speculative to enter this phase. The instruction may stall since the stores are sent to the memory hierarchy. The instruction may also get an exception (like a memory parity error). Retiring an instruction can be done out-of-order.

When committing an instruction the output values produced by the instruction are copied to the register bank. The instruction is automatically removed from the instruction tree, but it must be deallocated explicitly by calling ***SIM_instruction_end***. Instructions marked as speculative cannot be committed. This can be forced by the user by calling ***SIM_instruction_force_correct*** to remove the speculative status before committing. This is strongly discouraged since it may lead to incorrect execution. Also, the instruction may only have one valid execution path left, all alternative branches have to be squashed before the instruction can be committed.

Certain return values needs special actions to be taken. If some of the functions returns `Sim_IE_Exception` an exception occurred and the execution of the instruction is aborted. The exception should be handled by calling ***SIM_instruction_handle_exception*** on the faulting instruction. This call will set the program counter to the first instruction in the corresponding exception handler.

If `Sim_IE_Speculative` is returned it means a speculative instruction tried to commit.

If `Sim_IE_Sync_Instruction` is returned it means that a synchronizing instruction was executed in a speculative state. Synchronizing instructions have to be executed in non-speculative state since they cannot be squashed.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

`Sim_IE_OK` on success,
`Sim_IE_Code_Breakpoint` if a code breakpoint was encountered in the commit phase,
`Sim_IE_Step_Breakpoint` if a step breakpoint was encountered in the commit phase,
`Sim_IE_Exception` if the instruction raised an exception,
`Sim_IE_Unresolved_Dependencies` if the instruction are not allowed to execute due to unresolved dependences,
`Sim_IE_Speculative` if trying to commit a speculative instruction,
`Sim_IE_Stalling` if the instruction stalled on a memory access,
`Sim_IE_Sync_Instruction` if trying to execute a synchronizing instruction in an illegal way.

SIM_instruction_read_input_reg()

NAME

SIM_instruction_read_input_reg, SIM_instruction_read_output_reg, SIM_instruction_write_input_reg, SIM_instruction_write_output_reg — read/write register values

SYNOPSIS

```
attr_value_t
SIM_instruction_read_input_reg(instruction_id_t ii,
                               register_id_t ri);


attr_value_t
SIM_instruction_read_output_reg(instruction_id_t ii,
                                register_id_t ri);


void
SIM_instruction_write_input_reg(instruction_id_t ii,
                                register_id_t ri,
                                attr_value_t val);


instruction_error_t
SIM_instruction_write_output_reg(instruction_id_t ii,
                                 register_id_t ri,
                                 attr_value_t val);
```

DESCRIPTION

SIM_instruction_read_input_reg returns the value of an input register for instruction *ii* identified by the register id *ri*. The register id:s for a particular instruction can be retrieved by a calling ***SIM_instruction_get_reg_info***.

For integer and control registers the `attr_value_t` will be of kind `Sim_Val_Integer`. Floating point registers uses also the `Sim_Val_Integer` kind but the floating point number is stored as raw bits. Only single precision registers can be read, thus to read a double precision register both its low and high half must be read in separate calls. For quad precision registers four calls are required (SPARC).

`Sim_Val_Nil` will be returned if the value has not yet been produced by earlier instructions (see below).

The value is not read from the architecturally register state but from an internal renamed register which holds the input value of the corresponding register. Thus reading the same register from different instructions may yield different values even in the same cycle.

The value read is either produced by an previously executed instruction (this requires that the instructions are inserted in the tree and decoded) or explicitly set by the user

using ***SIM_instruction_write_input_reg***. In the later case the instruction will be regarded as speculative until an earlier non-speculative instruction produces the same value. If this never happens the instruction cannot be committed without the risk of executing incorrectly. This is how value prediction can be modeled (see the description of ***SIM_instruction_write_output_reg*** below as well).

SIM_instruction_proceed and ***SIM_instruction_commit*** will not accept a speculative instruction to commit. If this is requested the user needs to call ***SIM_instruction_force_correct*** first to redirect the correct execution path. This is however strongly discouraged since it may lead to incorrect execution.

SIM_instruction_read_output_reg reads the output value of a register from the instruction. This requires the instruction to be executed first.

SIM_instruction_write_output_reg writes the output register for an instruction. This is a convenient way of handling value speculation. The value written will propagate down the instruction tree exactly as if ***SIM_instruction_write_input_reg*** had been used for all instructions that use the value. This function can only be used for instructions that have not yet reached the execution phase.

EXCEPTIONS

Index Thrown if *ii* is illegal or if *ri* is an illegal register.

RETURN VALUE

`attr_value_t/Sim_Val_Integer` on success,
`attr_value_t/Sim_Val_Nil` if the value is not yet produced,
SIM_instruction_write_output_reg returns `Sim_IE_Illegal_Phase` if the instruction has reached the execution phase.

SEE ALSO

SIM_instruction_get_reg_info

SIM_instruction_remaining_stall_time()

NAME

SIM_instruction_remaining_stall_time — get stalling time

SYNOPSIS

```
cycles_t
SIM_instruction_remaining_stall_time(instruction_id_t ii);
```

DESCRIPTION

Returns the remaining time in cycles the instruction *ii* has to stall. Only applicable if ***SIM_instruction_status()*** `& Sim_IS_Stalling` is non-zero.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The stalling time in cycles.

SIM_instruction_rewind()

NAME

SIM_instruction_rewind — rewind phase of instructions

SYNOPSIS

```
instruction_error_t
SIM_instruction_rewind(instruction_id_t ii, instruction_phase_t phase);
```

DESCRIPTION

This function is used to undo the action taken by one or more phases of an instruction. Contrary to ***SIM_instruction_squash***, the instruction is not removed from the tree and deallocated. This can by useful if for example speculative data was used during a phase and it has been proved wrong later. It is then possible to rewind that phase and try again.

phase is the phase to rewind instruction *ii* to.

Currently the only phase that can be rewinded is the execution phase.

It is not possible to rewind a stalling instruction. The stalling must be finished before the instruction can be rewinded.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

Returns Sim_IE_OK on success, Sim_IE_Illegal_Rewind_Phase if this phase cannot be rewinded, or Sim_IE_Stalling if the instruction is stalling.

SIM_instruction_set_stop_phase()

NAME

SIM_instruction_set_stop_phase — set breakpoint on instruction phase

SYNOPSIS

```
void
SIM_instruction_set_stop_phase(conf_object_t *NOTNULL cpu,
                               instruction_phase_t phase,
                               int on);
```

DESCRIPTION

This function is used to instruct Simics on which instruction phases ***SIM_instruction_proceed*** should stop on. The defined phases are currently:

```
typedef enum instruction_phase {
        Sim_Phase_Initiated,
        Sim_Phase_Fetched,
        Sim_Phase_Decoded,
        Sim_Phase_Executed,
        Sim_Phase_Retired,
        Sim_Phase_Committed,
        Sim_Phases
} instruction_phase_t;
```

Thus ***SIM_instruction_set_stop_phase(cpu, Sim_Phase_Executed, 1)*** will cause every call to ***SIM_instruction_proceed(cpu, ii)*** to advance instruction *ii* to the point where its semantics has been executed.

SIM_instruction_speculative()

NAME

SIM_instruction_speculative — check if speculative instruction

SYNOPSIS

```
int
SIM_instruction_speculative(instruction_id_t ii);
```

DESCRIPTION

Note: this function was previously called ***SIM_instruction_wrong_path***.

This function check whether an instruction will execute or has executed using speculative input, i.e. the instruction may give incorrect result if committed. Speculative input can be used to model branch and value prediction, see ***SIM_instruction_write_input_reg*** and ***SIM_instruction_write_output_reg*** for a description on how to write speculative data.

ii is the id of the instruction to check,

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

Returns 1 for executing with speculative input, otherwise 0.

SIM_instruction_squash()

NAME

SIM_instruction_squash — rollback instructions

SYNOPSIS

```
instruction_error_t
SIM_instruction_squash(instruction_id_t ii);
```

DESCRIPTION

This function is used to rollback the action taken by an instruction and its children. This is needed for example when speculation has gone wrong and the state of the CPU must be reset to a point that is consistent with program order. This function will also call ***SIM_instruction_end*** for every instruction squashed to deallocate the data structures for the instruction.

If the hap Instruction_Squashed has been installed the associated callback will be called for each instruction before it is deallocated. This is useful for example if user data has been added to the instruction.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

Returns Sim_IE_OK on success, Sim_IE_Not_Inserted if the instruction is not inserted in the tree, Sim_IE_Sync_Instruction if there is a synchronous instruction in the tree that cannot be squashed, or Sim_IE_Retired_Instruction if there is a retired instruction in the tree.

SEE ALSO

SIM_instruction_rewind

SIM_instruction_stalling_mem_op()

NAME

SIM_instruction_stalling_mem_op — get stalling mem op

SYNOPSIS

```
generic_transaction_t *
SIM_instruction_stalling_mem_op(instruction_id_t ii);
```

DESCRIPTION

Returns the memory transaction that the instruction with id *ii* is currently stalling on. If there is no stalling transaction NULL is returned.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The stalling memory transaction or NULL if the instruction is not stalling.

SIM_instruction_status()

NAME

SIM_instruction_status — get instruction status

SYNOPSIS

```
instruction_status_t
SIM_instruction_status(instruction_id_t ii);
```

DESCRIPTION

Returns the instruction status of the instruction identified by *ii*. `instruction_status_t` is a bit field of the following values:

`Sim_IS_Ready` means that the instruction have all its input ready and can be executed (for the first time or after a stalling period), `Sim_IS_Waiting` means that the instruction is waiting for its dependences to be ready, `Sim_IS_Stalling` indicates that the instruction is stalling and does not have its output ready, `Sim_IS_Faulting` means that the instruction has raised an exception, `Sim_IS_Trap` on x86 means that the instruction has raised a trap, `Sim_IS_Interrupt` on x86 means that the interrupt received must be handled before executing further instructions, see SIM_instruction_phase), `Sim_IS_Branch_Taken` means that the branch was taken for a branch instruction.

If both `Sim_IS_Stalling` and `Sim_IS_Ready` are set it indicates that the instruction has completed its stalling period and needs to be executed (proceeded) again to make another call to the memory hierarchy, which this time presumably will "unstall" the instruction.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The status of the instruction in the instruction queue.

SIM_instruction_store_queue_mem_op()

NAME

SIM_instruction_store_queue_mem_op — get store queue transaction

SYNOPSIS

```
generic_transaction_t *
SIM_instruction_store_queue_mem_op(instruction_id_t ii, int i);
```

DESCRIPTION

Returns the *i*:th memory transaction that the instruction has put into the store queue. The value to be stored can be retrieved by calling ***SIM_get_mem_op_value*** on the returned transaction. If *i* is out of range, NULL will be returned.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The i:th memory transaction or NULL if there is no such transaction.

SIM_instruction_type()

NAME

SIM_instruction_type — return the instruction type

SYNOPSIS

```
instr_type_t
SIM_instruction_type(instruction_id_t ii);
```

DESCRIPTION

Returns the instruction type of the instruction identified by *ii* in the instruction queue. instr_type_t is a bit-field of the different instruction types. Currently only `It_Load`, `It_Store`, and `It_User_Decoder_Defined` are available. For instructions that swap data both `It_Load` and `It_Store` are set.

EXCEPTIONS

Index Thrown if *ii* is illegal.

RETURN VALUE

The type of the instruction in the instruction queue.

Appendix B

SPARC-V9 Instructions

Here follows a list of the SPARC instructions that cannot run out of order. The synchronous column tells if the instructions falls into category 1 or 2. Category 1 means that the instructions can only be executed if all earlier instructions in the instruction tree has been executed. Category 2 means that the instruction can only be executed if it is the only instruction in the tree. For comments see below.

Instruction	**synchronous**	**comment**
CASA		2
CASXA		2
DONE	2	
FLUSH	1	
FLUSHW	2	
ILLTRAP	2	1
LD(SB,SH,SW,UB,UH,UW,X)A		2
LDDA		2
LDDFA		2
LDFA		2
LDFSR	2	
LDQFA		2
LDSTUBA	2	
LDXFSR	2	
MEMBAR	2	
PREFETCHA		2
RDSTICK	1	
RDTICK	1	
RETRY	2	
ST(B,H,W,X)A		2
STBAR	2	
STDA		2

Instruction	synchronous	comment
STDFA		2
STFA		2
STFSR	2	
STQFA		2
STXFSR	2	
SWAPA		2
Tcc	2	1
WRASR	2	3
WRPR	2	4

Comments:

1. The instruction will cause an exception which makes it synchronous
2. If the ASI used by the instruction changes mappings in the MMU the instruction will be synchronous of category 2 otherwise it is not synchronous.
3. Writes to the registers %y, %ccr, %asi, %pc are NOT synchronous.
4. Writes to the registers %cansave, %canrestore, %otherwin, and %cleanwin are NOT synchronous.

Appendix C

x86 Instructions

Synchronized instructions of type 1:

ENTER	allocate stack space when entering procedure
POPA(D)	pop all registers from stack
PUSHA(D)	push all registers onto the stack

Synchronized instructions of type 2:

CALL	call procedure (in other seg.)
CLFLUSH	Flushes cache lines
CLI	clear interrupt flag
CLTS	clear task-switched flag in CR0
CPUID	returns processor identification information
EMMS	empty MMX state
FP instr.	
FXRSTOR	restore x87 FPU, MMX, XMM, and MXCSR state
HLT	halt
IN	input from port; fixed port
IN	nput from port; variable port
(REP) INS	input from DX port
INT1	
INT n	interrupt type n
INTO	interrupt 4 on overflow
INT	single step interrupt 3
INVD	invalidate cache
INVLPG	Invalidate TLB Entry
IRET/IRETD	interrupt return
JMP	unconditional jump (to other seg.)
JMP	unconditional jump (to other seg.)
LDMXCSR	Load MXCSR

LDS	load pointer to DS
LES	load pointer to ES
LFENCE	Serializes load operations
LFS	load pointer to FS
LGDT	Load Global Descriptor Table Register
LGS	load pointer to GS
LLDT	Load Local Descriptor Table Register
LMSW	Load Machine Status Word
LOCK prefixed instr.	
LSS	load pointer to SS
LTR	Load Task Register
MFENCE	Serializes load and store operations
MOV	data to segment register
MOV	move to control register
MOV	move to debug register
OUT	output to port; fixed port
OUT	output to port; variable port
OUTS	output to DX port
POP	pop top of stack into DS
POP	pop top of stack into FS
POP	pop top of stack into GS
PUSH	push CS onto the stack
PUSH	push DS word onto the stack
PUSH	push ES onto stack
PUSH	push SS onto stack
RET	return from procedure (to other seg)
RSM	resume from system management mode
SFENCE	Serializes store operations
STI	set interrupt flag
SYSENTER	fast call to pl 0 system procedures
SYSEXIT	fast return from fast system call
WAIT	wait
WBINVD	write-back and invalidate data cache
WRMSR	write to model-specific register

Index

T

V

W

Virtutech, Inc.

1740 Technology Dr., suite 460
San Jose, CA 95110
USA

Phone +1 408-392-9150
Fax +1 408-608-0430

http://www.virtutech.com