ADL ISS Model Integration

This document provides an overview of how to integrate an ADL-generated ISS library into another application, such as a system model.

Author: Brian Kahne
Contact: bkahne@freescale.com

Table of Contents

1 General Overview

An ADL-generated ISS model can be used in multiple ways: If linked with the appropriate libraries it can function as a stand-alone executable or, without these libraries, can be embedded into another application, such as a model of a system. This document describes the latter use and provides an overview of the API used to communicate with the model.

This API is designed to be as simple and relatively low-level as possible, since the model might be used in a relatively wide range of applications. For example, the logging and tracing mechanism consists of a set of static functions which are called whenever an event of interest occurs. These functions purposefully avoid any kind of heap allocation; it is up to the external application to store this data in a container if necessary. Likewise, only the current state of memory is stored. It is up to the external application to track what addresses have been used, if this information is important.

2 Building an ADL Library

The main tool used to generate an ISS from an ADL source file is `adl2iss`. To create a base library, the `--target=base-so` option is used. For example:

```
adl2iss model.adl --output=model-base.so --target=base-so
```

This reads `model.adl` as input and produces `model-base.so`, a shared library which contains the ISS and is linked to the necessary support libraries.

If additional capabilities, such as initializing the model from an input file and logging simulation activity to a trace file are required, then the `--target=so` option may be used, which links the generated model with additional libraries. This added functionality is not covered by this document.

By default, the library that is generated depends upon the `RNumber` library, which is a dynamic, arbitrary-sized integer class. This class is used by the ISS debug API to provide support for getting and setting register values of arbitrarily-large width. However, it is possible to disable this support by using the `--no-rnumber` option. If this option is used, then the generated library will no longer be dependent upon RNumber. Instead, all register debug accesses will use a 64-bit integer instead, meaning that it will not be possible to access registers which are wider than 64-bits.

By default, the generated ISS is contained within the namespace `adliss`. If more than one generated simulator is to be included into a single application, then each simulator must have a unique namespace. This can be done by using the `iss-namespace=<name>` option.

Various other options may also be specified, such as to disable tracing or debug support. Run `adl2iss --help` to list all available options.

3 The ADL ISS API

The ADL ISS API is declared in the header file `ModelInterface.h`. Functions for controlling the simulation are declared in `SimInterface.h`. In a normal installation,

these header files will be installed in `<prefix>/include/iss`, where `<prefix>` is the base path of the ADL installation.

Note that if RNumber support is not included then you must define `__NO_RNUMBER__` before including `ModelInterface.h`. Also note that all functions are contained within the `adl` namespace.

All core models are derived from `IssCore` while systems of cores are derived from `IssSystem`. Both of these classes are derived from `IssNode`. This class provides various methods for getting and setting various resource values, an introspection mechanism to allow for querying the model for details of the resources it contains, and simulation control.

3.1 Instantiating the Model

All ADL ISS models are re-entrant. A given model library has a single top-level model element which might be a system containing multiple cores or systems of cores. Only the top-level element may be explicitly instantiated by an external application.

The function `IssNode *adliss::createTopLevelNode(unsigned &id)` instantiates this top-level element and returns a pointer to the object. This function may be called multiple times in order to create multiple instances of the model. The `id` parameter is used to assign a unique ID to each core that is allocated. Each core uses the given value and then post-increments it.

If a model is placed into a unique namespace, then you must change `adliss` to the appropriate string. In order to get the declaration for this function, `ModelInterface.h` may be included multiple times with different defines for `ISS_NAMESPACE`. For example:

```
#define ISS_NAMESPACE foo
#include ModelInterface.h

#define ISS_NAMESPACE bar
#include ModelInterface.h
```

In this example, the namespaces `foo` and `bar` were used to differentiate between two different generated simulators.

3.2 Simulation Control

An ADL ISS model may be reset on a per-core basis by calling `IssNode::reset()`. This resets a model to its power-on-reset values. All global resources, such as memory, may also be reset by calling `resetGlobal()`.

The following API functions exist for controlling the model during simulation:

- `ProgramStatus exec_from_buffer(uint32_t *buf,unsigned n)`: Directly executes an instruction stored in `buf`, where n specifies the number of words in `buf`. Data is copied from `buf` to an internal buffer, then executed, skipping the normal fetch sequence. This is useful for modeling features such as an external-debug mode, where an outside controller might insert instructions directly into the core.
- `void run()`: This cycles the model until there are no more active cores. Execution may be interrupted by throwing a `SimInterrupt` exception, explained below.
- `ProgramStatus step()`: Cycles the model once. If this is a system, then all constituent items are cycled once. Returns a status value, declared in `ModelInterface.h`, which describes the resulting state of the model, e.g. if the model is active, hit a breakpoint or watchpoint, or is halted. Refer to log_breakpoint for more information about breakpoints.
- `ProgramStatus stepn(unsigned n)`: Cycles the model n times. If this is a system, then all constituent items are cycled n times. Returns a status value, declared in `ModelInterface.h`, which describes the resulting state of the model, e.g. if the model is active, hit a breakpoint or watchpoint, or is halted.

If the model was generated with tracing and debug support then breakpoints and watchpoints may be set by an external application. The API for doing this is:

- `unsigned setBreakpoint(addr_t,PtBase *payload = 0)`: Set a breakpoint at the specified effective address. Returns an integer handle for the breakpoint.
- `unsigned setTmpBreakpoint(addr_t,PtBase *payload = 0)`: Set a temporary breakpoint at the specified effective address. This type of breakpoint will be erased once it has been triggered. Returns an integer handle for the breakpoint.
- `unsigned setWatchpoint(addr_t,WatchType type,PtBase *payload = 0)`: Set a watchpoint at the specified effective address. `type` is an enumerated type describing the types of actions to which the watchpoint is sensitive, e.g. on a read or a write, and is declared in `BasicTypes.h`. Returns an integer handle for the watchpoint.
- `void clearBreakpoints()`: Clear all breakpoints in the model.
- `void clearWatchpoints()`: Clear all watchpoints in the model.
- `bool clearBreakpoint(unsigned h)`: Clear a breakpoint, specified by a given handle. Returns true on success, false on failure.
- `bool clearWatchpoint(unsigned h)`: Clear a watchpoint, specified by a given handle. Returns true on success, false on failure.
- `bool clearBreakpointByAddr(addr_t addr)`: Clear a breakpoint, specified by an address. Returns true on success, false on failure.
- `bool clearWatchpointByAddr(addr_t addr,WatchType type)`: Clear a watchpoint, specified by an address and type. Returns true on success, false on failure.

When a breakpoint or watchpoint is encountered, the model will call the respective logging function, `log_breakpoint()` or `log_watchpoint`, with the handle of the breakpoint or watchpoint and the payload pointer that was provided.. These functions are described in more detail in The Logging API. The call-back function may then throw a `SimInterrupt` exception in order to stop the simulation.

3.3 The Debug API

The debug API allows an external program to query the present state of the model and to modify internal resources. Various methods of `IssNode` are used to perform these actions. This section describes only a portion of the API; please refer to `ModelInterface.h` for a complete description. In general, *get* and *set* routines exist for various types of resources, enabling an external application to read and write resource values.

In addition, *show* routines provide a mechanism for iterating over all resources of a given type. A *show* routine takes a reference to a `ReportBase` object and calls a relevant method of this object for each resource of a given type in the model. For example, `ReportBase::report_req` is called for each register when `showRegs` is executed.

3.3.1 Register Modification

- `bool setReg(const std::string &name,unsigned index,uint64 value)`: Sets a register of the specified name. Returns false if the specified register does not exist.

  Arguments:

  name
  : The name of the register to modify.

  index
  : If this is a register file, **index** may be used to specify an element of the register file. An element of a register file may also be modified by appending the index to its name, e.g. `GPR5` would modify element 5 of a register file named `GPR`.

  value:
  : The value to which the register should be set. Note that this function does not cause side-effects specified using register write hooks. However, the state of the model is updated, such as for activating contexts.

- `bool setReg(const std::string &name,unsigned index,const rnumber::RNumber &value)`: Same as above. However, **value** is an RNumber, which allows a register of arbitrary width to be set. This function is only present if the model was generated with RNumber support.

- `bool getReg(const std::string &name,unsigned index,uint64 &value) const`: Get the value of a register. Returns false if the register does not exist.

  Arguments:

  name
  : The name of the register.

  index
  : If this is a register file, **index** may be used to specify an element of the register file. An element of a register file may also be read by appending the index to its name, e.g. `GPR5` would retrieve element 5 of a register file named `GPR`.

  value
  : The value of the register is stored into this parameter.
- `bool getReg(const std::string &name,unsigned index,rnumber::RNumber &value) const`: Same as above. However, **value** is an RNumber, which allows a register of arbitrary width to be read. This function is only present if the model was generated with RNumber support.
- `void showRegs(ReportBase &) const`: Causes the model to iterate through all registers and register-files, calling the `report_reg` and `report_regfile` methods of `ReportBase` for each element. The `ReportBase` class is an interface and is declared in `ModelInterface.h`.

3.3.2 Register and Exception Callbacks

ISS models support register and exception callbacks: The ability to notify a registered functor when a value has changed or when an exception has been raised. At model-build time, the user must specify which registers or exceptions should have callback support. This is done via a config file. For example, given an ADL model with a core named `P`, the following code (placed into a configuration file named, for example `config.ttc`) would specify that registers `HID0` and `HID1` should have callback support:

```
define(core=P) {
  define (config) {
    reg_callbacks = (HID0,HID1);
  }
}
```

For exceptions, the key is `exception_callbacks`. This file would then be specified on the `adl2iss` command-line using the following option: `--config-file=config.ttc`.

A platform model's ADL wrapper could then register callbacks by calling `IssNode::setRegCallback`. Two versions of this function exist:

```
virtual bool setRegCallback(unsigned rindex,RegCallback *cb);
virtual bool setRegCallback(const std::string &name,RegCallback *cb);
```

The first version takes an integer handle to specify the register; the second uses the register's name as a string.

For exceptions, the registration functions are:

```
virtual bool setExeptCallback(unsigned index,ExceptCallback *cb);
virtual bool setExceptCallback(const std::string &name,ExceptCallback *cb);
```

A callback is simply an object derived from *RegCallback* or *ExceptCallback*:

```
struct RegCallback {
  virtual ~RegCallback() {};
  virtual void operator()(unsigned index,REGTYPE value) {};
};

struct ExceptCallback() {
  virtual ~ExceptCallback() {};
  virtual void operator()() {};
};
```

On an update of the specified register, `operator()` is called with the new value. In the case of a register-file, the index of the update is also called. ADL does not currently support callbacks for specific elements of a register-file. For an exception, `operator()` is called immediately after the exception's action code has been executed.

For register callbacks, the value's type, `REGTYPE`, depends upon how the model is built. Normally, it will be of type `RNumber`, but if the model is built without RNumber support, then it will be of type `uint64_t`. In order to be consistent, wrapper code which includes `ModelInterface.h` must define `__NO_RNUMBER__` if the model was built with no RNumber support.

3.3.3 Memory Modification

- `void setMem(const std::string &name, addr_t addr, uint32 data, unsigned size)`: Set a value in memory.

  Arguments:

  name
  : The name of the memory. The global memory is named **Mem**.

  addr
  : The destination address. This is a real address; no translation is performed.

  data
  : The new memory data value.

  size
  : The number of bytes (1 to 4) to actually write.

- `uint32 getMem(const std::string &name, addr_t addr)`: Read memory.

  name
  The name of the memory. The global memory is named **Mem**.
  addr
  The source address. This is a real address; no translation is performed.
- `uint32 getMem(unsigned id, addr_t addr)`: Same as above, except that this version allows the user to specify the source memory using an integer handle in order to avoid the overhead of string comparisons. The global memory always has an index of 0.
- `void showMem(ReportBase &)`: Causes the model to iterate through all registers and register-files, calling the `report_memory` method of ``ReportBase` for each element. The `ReportBase` class is an interface and is declared in `ModelInterface.h`. The model does not track what memory is, or is not, initialized. This must be done by an external object if such information is necessary.

The reason that each `IssNode` object contains an API for reading and writing memory is that it is possible to define memories which are local resources of the core. In the case where such memories do not exist and there is only a single global memory, this API can be bypassed. The following routines can be used to read and write memory globally:

- `uint64 mem_read64 (addr_t addr)`: Read a 64-bit value from the global memory.
- `uint32 mem_read32 (addr_t addr)`: Read a 32-bit value from the global memory.
- `uint16 mem_read16(addr_t addr)`: Read a 16-bit value from the global memory.
- `uint8 mem_read8 (addr_t addr)`: Read an 8-bit value from the global memory.
- `void mem_write64 (addr_t addr,uint64 value,uint64 mask=0xffffffffffffffffULL)`: Write a 64-bit value to memory. The mask specifies what bits of `value` are written to memory. The default is to write the entire 64-bit value.
- `void mem_write32 (addr_t addr,uint32 value,uint32 mask=0xffffffff)`: Write a 32-bit value to memory.
- `void mem_write16(addr_t addr,uint16 value,uint16 mask=0xffff)`: Write a 16-bit value to memory.
- `void mem_write8 (addr_t addr,uint8 value)`: Write an 8-bit value to memory.

As noted above, all addresses are real addresses and no address translation is performed. If it is necessary to translate addresses then the translation must

be performed explicitly using the following API. Note that these are methods of `IssCore`, so it may be necessary to perform a `dynamic_cast` in order to obtain a pointer of the necessary type. These translations are meant to not cause side-effects: Permission checking or miss handling are not performed.

- `bool extInstrReadTranslate(addr_t &ra,addr_t ea,bool log = false)`: Translate an effective address to a real address as an instruction-read, using the core's MMU. If `log` is true, then MMU logging events are generated, even if normal tracing for this type of event is turned off.
- `bool extDataReadTranslate(addr_t &ra,addr_t ea,bool log = false)`: Translate an effective address to a real address as a data-read, using the core's MMU. If `log` is true, then MMU logging events are generated, even if normal tracing for this type of event is turned off.
- `bool extDataWriteTranslate(addr_t &ra,addr_t ea,bool log = false)`: Translate an effective address to a real address as a data-write, using the core's MMU. If `log` is true, then MMU logging events are generated, even if normal tracing for this type of event is turned off.

The memory access functions directly read from and write to the memories, bypassing any caches which might be present. In order to read or write data from the top of the memory hierarchy, the following routines are provided. In the case of reads, the caches are not modified; if an address is not present, the routine goes to the next level of the memory hierarchy. In the case of writes, the routine visits each level of the memory hierarchy, updating the data if the address is already present, but not modifying the cache if it is not present.

- `void debug_instr_read(uint32_t &result,bool trans,addr_t addr,int size)`: Reads from the instruction memory hierarchy.
- `void debug_instr_write(bool trans,addr_t addr,uint32_t value,int size)`: Writes to the instruction memory hierarchy.
- `bool debug_data_read(uint32_t &result,unsigned id,bool trans,addr_t addr,int size)`: Reads from the data hierarchy. A specific memory may be designated via `id`; 0 represents the global memory.
- `bool debug_data_write(unsigned id,bool trans,addr_t addr,uint32_t value,int size)`: Writes to the data hierarchy. A specific memory may be designated via `id`; 0 represents the global memory.

3.4 The Introspection API

The introspection API allows an external program to query the model for information about the resources it contains. These methods are all members of `IssNode` and return constant data structures declared in `Introspection.h`.

- `const MmuInfo &getMmuInfo() const`: Return information about the model's MMU.

- `const RegInfos &getRegInfo() const`: Return information about the model's registers and register-files.
- `const CacheInfos &getCacheInfo() const`: Return information about the model's caches.
- `const MemInfos &getMemInfo() const`: Return information about the model's memories.
- `const ExceptionInfos &getExceptionInfo() const`: Return information about the model's exceptions.
- `const CtxInfos &getContextInfo() const`: Return information about the model's contexts.

3.5 The Logging API

If tracing is enabled, then the model will call logging methods of its installed logger object when various actions occur, such as when a register is written or memory is accessed. The logging interface is defined in `ModelInterface.h` and is called `LogBase`.

If the model has been generated with tracing support then the logging of different types of events can be controlled on an individual basis. Event types are described using the enumerated type `TraceType`, declared in `ModelInterface.h`. The API for turning tracing on and off is:

- `unsigned set_tracing(unsigned flags = (unsigned)-1)`: Toggle on tracing. The value is or'd with the current value and the previous value is returned. The default is to activate all logging.
- `unsigned clear_tracing(unsigned flags = (unsigned)-1)`: Toggle off tracing. The bits set in the argument turn off the relevant tracing activities. Default is to turn off all events. Returns the prior value.
- `unsigned set_tracing_flags(unsigned flags)`: Set the tracing flags directly. Returns the prior value.
- `bool tracing_on()`: Returns true if any tracing is activated.
- `bool tracing_on(unsigned flags)`: Returns true if the specified tracing events are all on.
- `unsigned tracing_flags()`: Returns the current tracing flags.

The `IssNode` interface for installing and accessing logging objects is:

- `LogBase &IssNode::logger() const`: Return a reference to the installed logger. Some sort of logger is always installed, even if it is just a place-holder which does nothing.
- `void IssNode::setLogger(LogBase *)`: Install a new logger. If the argument is 0, then a dummy logger is set instead. Although this interface is defined for `IssNode`, only `IssCore` objects actually implement the interface.

The following are the event logging methods of `LogBase`:

- `void log_instr_prefetch(addr_t ea)`: Logs the fact that an instruction is going to be fetched.

  Arguments:

  ea
  : The effective address of the impending fetch.
- `void log_instr_read(unsigned id,const char *name,addr_t ea,addr_t ra,uint32 value)`: Logs the instruction fetch action itself.

  Arguments:

  id
  : Numerical identifier for the memory.

  name
  : Memory from which the instruction is being read.

  ea/ra
  : Effective and real address of the read.

  value
  : The memory value that was read.
- `void log_instr(uint32 *opc,int num_half_bytes,const char *name,Disassembler dis,uint32_t flags)`: Logs the instruction usage.

  Arguments:

  opc
  : The opcode of the instruction.

  num_bytes
  : The length of the instruction in bytes.

  name
  : The name of the instruction.

  dis
  : A disassembler function, which will write a complete disassembly line to a supplied stream.

  flags
  : Information about the instruction. The bit flags are defined by **InstrFlags** in **ModelInterface.h**.
- `void log_instr_issue_time(ttime_t issue_time)`: For a time-tagged ISS, this logs the issue time of the instruction.

  Arguments:

  issue_time
  : Time when the instruction was issued.

- `void log_instr_completion_time(ttime_t issue_time)`: For a time-tagged ISS, this logs the completion time of the instruction.

  Arguments:

  completion_time
  : Time when the instruction was issued.
- `void log_reg_write(unsigned id,const char *name,REGTYPE value)`: Logs a register write operation.

  Arguments:

  id
  : Numerical identifier for the register.

  name
  : Name of the register.

  value
  : The new value of the register. The type of this parameter is `uint64` if the model was generated with `--no-rnumber` and `const rnumber::RNumber &` otherwise.
- `void log_regfile_write(unsigned id,const char *name,uint32 index,REGTYPE value)`: Logs a register-file write operation.

  id
  : Numerical identifier for the register file.

  name
  : Name of the register.

  index
  : Index of the element being written.

  value
  : The new value of the register. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.
- `void log_reg_read(unsigned id,const char *name,REGTYPE value)`: Logs a register read operation. This is only called if the model was generated with **--log-reg-reads**.

  Arguments:

  id
  : Numerical identifier for the register.

  name
  : Name of the register.

value
: The new value of the register. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.

- `void log_regfile_read(unsigned id,const char *name,uint32 index,REGTYPE value)`: Logs a register-file read operation. This is only called if the model was generated with **--log-reg-reads**.

  id
  : Numerical identifier for the register file.

  name
  : Name of the register.

  index
  : Index of the element being written.

  value
  : The new value of the register. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.

- `void log_reg_write_mask(unsigned id,const char *name,REGTYPE value,REGTYPE mask)`: Logs a partial register write operation. This is only called if the model was generated with the **--log-reg-masks** option.

  Arguments:

  id
  : Numerical identifier for the register.

  name
  : Name of the register.

  value
  : The new value of the register. The type of this parameter is **uint64** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.

  mask
  : The portion of the register accessed. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.

- `void log_regfile_write_mask(unsigned id,const char *name,uint32 index,REGTYPE value,REGTYPE mask)`: Logs a partial register-file write operation. This is only called if the model was generated with the **--log-reg-masks** option.

  id
  : Numerical identifier for the register file.

name
Name of the register.
index
Index of the element being written.
value
The new value of the register. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.
mask
The portion of the register accessed. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.

- `void log_reg_read_mask(unsigned id,const char *name,REGTYPE value,REGTYPE mask)`: Logs a partial register read operation. This is only called if the model was generated with the **--log-reg-reads** and **--log-reg-masks** options.

  Arguments:

  id
  Numerical identifier for the register.
  name
  Name of the register.
  value
  The new value of the register. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.
  mask
  The portion of the register accessed. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.
- `void log_regfile_read_mask(unsigned id,const char *name,uint32 index,REGTYPE value,REGTYPE mask)`: Logs a register-file read operation. This is only called if the model was generated with the **--log-reg-reads** and **--log-reg-masks** options.

  id
  Numerical identifier for the register file.
  name
  Name of the register.
  index
  Index of the element being written.

value
The new value of the register. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.
mask
The portion of the register accessed. The type of this parameter is **uint64_t** if the model was generated with **--no-rnumber** and **const rnumber::RNumber &** otherwise.

- `void log_core_mem_write(unsigned id,const char *name,addr_t ea,int num_bytes)`: Logs a memory write from the core's point of view.

  id
  Numerical identifier for the memory.
  name
  Name of the memory.
  ea
  Effective address of the operation.
  num_bytes
  Number of bytes of the operation.
- `void log_core_mem_read(unsigned id,const char *name,addr_t ea,int num_bytes)`: Logs a memory read from the core's point of view.

  id
  Numerical identifier for the memory.
  name
  Name of the memory.
  ea
  Effective address of the operation.
  num_bytes
  Number of bytes of the operation.
- `void log_core_mem_write_typed(unsigned id,const char *name,addr_t ea,int num_bytes,CacheAccess type)`: Logs a memory write from the core's point of view. This is used when a *typed* access is performed, i.e. the user has specified a specific CacheAccess value for the memory access.

  id
  Numerical identifier for the memory.
  name
  Name of the memory.
  ea
  Effective address of the operation.
  num_bytes
  Number of bytes of the operation.

type
The user-specified access type.

- `void log_core_mem_read_typed(unsigned id,const char *name,addr_t ea,int num_bytes,CacheAccess type)`: Logs a memory read from the core's point of view. This is used when a *typed* access is performed, i.e. the user has specified a specific CacheAccess value for the memory access.

  id
  Numerical identifier for the memory.
  name
  Name of the memory.
  ea
  Effective address of the operation.
  num_bytes
  Number of bytes of the operation.
  type
  The user-specified access type.
- `void log_mem_write(unsigned id,const char *name,bool pre,int seq,addr_t ea,addr_t ra,uint32 value)`: This reflects a memory write from the point of view of the memory's new state.

  id
  Numerical identifier for the memory.
  name
  Name of the memory.
  pre
  If true, the call is before the action is taken. If false, the action has occurred and the value reflects the new state of memory.
  seq
  This indicates which write is occurring. This will only be non-zero for misaligned writes.
  ea/ra
  Effective and real address of the operation. This is always a 32-bit word-aligned address.
  value
  The new value of memory.
- `void log_mem_read(unsigned id,const char *name,bool pre,int seq,addr_t ea,addr_t ra,uint32 value)`: This reflects a memory read from the point of view of the memory's new state.

  id
  Numerical identifier for the memory.
  name
  Name of the memory.

pre
: If true, the call is before the action is taken. If false, the action has occurred and the value reflects the new state of memory.

seq
: This indicates which write is occurring. This will only be non-zero for misaligned writes.

ea/ra
: Effective and real address of the operation. This is always a 32-bit word-aligned address.

value
: The value of memory that was read.

- `void log_mmu_translation(TransType tt,int seq,int set,int way,addr_t ea,addr_t ra,const MmuBase *mb)`: Logs a memory-translation operation. Note that this is called whenever an address translation hits within a TLB, regardless of whether the translation is legal according to the various permission checks.

  tt
  : Type of translation being performed. This enumerated type is declared in the header file `BasicTypes.h`.

  seq
  : Sequence number for misaligned accesses

  set
  : The index of the affected set. Only valid for writes to the array; not currently reported for lookups.

  way
  : The index of the affected way. Only valid for writes to the array; not currently reported for lookups.

  ea
  : The effective address which initiated the translation. For `tt=WriteTrans`, the address is invalid and will always be 0.

  ra
  : The real address result of the translation. For `tt=WriteTrans`, the address is invalid and will always be 0.

  mb
  : A pointer to the translation object that was picked. This class is declared in **ModelInterface.h**. This pointer may refer to an entry in a TLB, in which the object pointed to is relatively long-lived, or may refer to a temporary object logged by the user, in which case the item will be very short-lived. Thus, if it is necessary to cache the object for longer than the lifetime of the logging function, then the object must be tested by calling `MmuBase::user_entry()`. If the result is true, then the object must be cloned by calling `MmuBase::clone()`.

- `void log_cache_action(const char *name,CacheAction action,CacheAccess access,unsigned level,int set,int way,unsigned linemask,addr_t ra)`: Logs a cache access/operation.

  name
  : The name of the cache.

  action
  : The action type, e.g. hit, miss, evict, etc. This enumerated type is declared in **BasicTypes.h**.

  access
  : The type of access generating the action, e.g. instruction-fetch, data-read, etc. This enumerated type is declared in **BasicTypes.h**.

  level
  : Level in the hierarchy of caches.

  set/way
  : The set/way of the access. -1 for set or way values indicates invalid values that should be ignored.

  linemask
  : The line-mask value for this cache.

  ra
  : Real address of the cache access.
- `void log_breakpoint(addr_t ea,unsigned handle,PtBase *payload)`: This logs the occurrence of a breakpoint. This function is only relevant when debug support is enabled when generating the model.

  Breakpoints and watchpoints do not halt the model by default. Instead, a function, such as **log_breakpoint** must throw a **SimInterrupt** object. The model's simulation loop will catch this and return the appropriate **ProgramStatus** value.

  ea
  : Effective address of the access generating the break/watchpoint.

  handle
  : Integer handle of the break/watchpoint.

  payload
  : A pointer to a `PtBase` payload object that was supplied during the call to `setBreakpoint` or `setTmpBreakpoint`. Note that if this is a temporary breakpoint, it will be erased after the call to this logging function.
- `void log_watchpoint(addr_t ea,bool pre,WatchType type,unsigned handle,PtBase *payload)`: This logs the occurrence of a watchpoint. This function is only relevant when debug support is enabled when generating the model. Refer to log_breakpoint for information about how to stop a model when a watchpoint occurs.

Note that integration code normally does not want to throw a `SimInterrupt` within the watchpoint logging function because this will interrupt the instruction before it has completed. Instead, set a temporary breakpoint at the program-counter location and store relevant watchpoint information within the payload. This allows the instruction to complete before stopping the simulation.

ea
: Effective address of the access generating the break/watchpoint.

pre
: If true, the call is before the action (load or store) is taken. If false, the action has occurred.

handle
: Integer handle of the break/watchpoint.

type
: Type of watchpoint (load, store, etc.).

payload
: A pointer to a `PtBase` payload object that was supplied during the call to `setBreakpoint` or `setTmpBreakpoint`. Note that if this is a temporary breakpoint, it will be erased after the call to this logging function.

- `void log_exception(unsigned handle,bool pre,const char *name)`: Logs the occurrence of an exception.

  handle
  : Integer handle of the exception. This can be obtained by calling **IssNode::getExceptionInfo()** on a core.

  pre
  : If true, the call is before the exception has been taken. If false, the exception has occurred and all side-effects have occurred.

  name
  : The exception's name.

- `void log_annotation(MsgType type,unsigned level,const std::string &msg,unsigned ndata,Data data[])`: Logs the occurrence of an annotation message. This function will be called even if tracing is off, for warnings and errors.

  type
  : The type of message: *mInfo*, *mWarn*, or *mError*.

  level
  : User-defined annotation level.

  msg
  : The annotation message.

  ndata
  : The number of data arguments.

data

An array of data arguments, if any, specified by the user. The *Data* structure is:

```
struct Data {
  std::string _key;
  uint64_t    _value;
};
```

- `void log_branch_taken(addr_t ea)`: Logs the fact that an instruction was considered to be a taken branch. This corresponds to an update of the next-instruction-address register.

  ea
  Branch target effective address.
- `void log_core_change()`: Called when a core change is occurring in the middle of an instruction's intermediate result list, e.g. one core modifies another core's resources directly.
- `void log_ctx_change(unsigned id,const char *name,unsigned context_num,CtxType update)`: Called whenever a context change occurs.

  id
  Numerical identifier for the context.
  name
  The context's name.
  context_num
  The new active index.
  update

  This enumerated type describes the update type:

  ctxSwitch
  This is a general context switch event.
  ctxUpdate
  This is a temporary change due to an update to a resource in a specific context.
  ctxDone
  This signifies an end to the temporary update.

The events which are logged may be controlled individually, on a per-core basis. The control API is defined in `IssNode` and contains two forms: A non-recursive set of methods which act only upon the current node and a recursive form, which act upon all children in the hierarchy. Events are specified via an enum called `TraceType`, defined in `ModelInterface.h`.

The API is:

- **uint64_t set_tracing(uint64_t flags)**: Toggle on tracing. The value is or'd with the current value and the previous value is returned. The TraceType enum describes the different types of tracing events. The default is to set all values.
- **uint64_t clear_tracing(uint64_t flags)**: Toggle off tracing. The bits set in the argument turn off the relevant tracing activities. Default is to turn off all events. Returns the prior value.
- **uint64_t set_tracing_flags(uint64_t flags)**: Set the tracing flags directly. Returns the prior value.
- **bool tracing_on() const**: Returns true if any tracing is activated.
- **bool tracing_on(uint64_t flags) const**: Returns true if any tracing is activated.
- **uint64_t tracing_flags() const**: Returns the current tracing flags.
- **void set_tracing_r(uint64_t flags)**: Recursively sets the specified flags in all child nodes.
- **void clear_tracing_r(uint64_t flags)**: Recursively clears the specified flags in all child nodes.
- **void set_tracing_flags_r(uint64_t flags)**: Recursively sets the tracing flags to the value specified.
- **bool tracing_on_r() const**: Returns true if any child nodes have tracing enabled for any events.
- **bool tracing_on_r(uint64_t flags) const**: Returns true if any child nodes have tracing enabled for the events specified by `flag`.

3.6 Data Dependency Tracking API

When enabled, an ADL ISS has the capability to correlate resource reads with resource updates, both for registers and for memory. This data can then be queried by the user application, in order, or example, to track undefined data through the core.

In order for this feature to be enabled, the model must be built with the `--log-reg-reads` and `--dep-tracking` options enabled. The model must also contain `dependencies` blocks for all relevant instructions. Currently, this feature is only enabled for normal, interpreter-based ISSs.

The **IssNode** API for dependency tracking is:

- **bool has_dependency_tracking() const**: Returns true if the model supports dependency tracking.
- **unsigned get_reg_seq_num() const**: Returns the current register sequence number.
- **unsigned get_mem_seq_num() const**: Returns the current memory sequence number.
- **const DepItems *get_last_dependencies() const**: Returns the last dependency list which was selected by a write operation. If the list is empty, then no dependency information was recorded.

The **DepItems** data structure is:

```
struct DepItem {
  enum Type { dpReg, dpMem };

  DepItem(Type t,unsigned id,unsigned snum) : _type(t), _id(id), _snum(snum) {};

  Type     _type;  // Type for this item:  register or memory.
  unsigned _id;    // Id for item.
  unsigned _snum;  // Sequence number for this access.  Only valid for sources.
};

// Items in the vector are the sources.  Target information is stored in _trg.
struct DepItems : public std::list<DepItem> {};
```

The basic idea is that, for each register or memory logging event (**log_reg_read**, **log_regfile_read**, **log_reg_read_mask**, **log_regfile_read_mask**, **log_core_mem_read**), the user application should query for the current register or memory sequence number by calling **IssNode::get_reg_seq_num()** or **IssNode::get_mem_seq_num()**. Then, on a write logging event (**log_reg_write**, **log_regfile_write**, **log_reg_read_mask**, **log_regfile_read_mask**, **log_core_mem_write**), the user application should call **IssNode::get_last_dependencies()** to retrieve the dependency list for that write operation.

Each dependency item is a **DepItem** and contains a type specifier (register or memory), an id (useful for debugging, but not strictly necessary), and a sequence number, which is used to correlate with a previous read operation.

3.7 Disassembler API

A model may be generated with the `--disassembler` option, which will create a standalone disassembler within the model. This allows a user to disassemble arbitrary memory.

The disassembler function is a method of `IssNode`:

```
// o:      Disassembly output stream
// addr:   Source address.  This is an ea if tran=true, else an ra.
// tran:   Whether to translate.
// tindex: Index of instruction table to use, or -1 to use the current.
virtual unsigned disassemble(std::ostream &o,addr_t addr,bool tran,int tindex) const;
```

Memory at `addr`, where `addr` is either a real or effective (virtual) address, as determined by the `tran` parameter, is disassembled, and the textual output is sent to the output stream `o`. By default (a value of -1 for `tindex`) the model's current instruction table is used. This may be overridden by supplying a different instruction table index. These values may be obtained by retrieving the attribute information via a call to `adl::getAttrData()`.

3.8 Dynamic Parameter API

A model may be generated in such a way that various resources may be modified at startup-time. For example, caches can be configured so that their size may be set dynamically. An API is provided for modifying these parameters.

This API consists of methods in `IssNode`. The methods always exist, but whether any parameters may be set is dependent upon how the model was built

- `virtual void set_dyn_parm(const std::string &parm,unsigned value)`: Sets a parameter value. Throws a **runtime_error** if the parameter or value are invalid.
- `virtual unsigned get_dyn_parm(const std::string &parm) const`: Retrieves the value of a dynamic parameter. Throws a **runtime_error** if the parameter is invalid.
- `virtual void list_dyn_parm(StrPairs &parms) const`: Returns a sequence of pairs describing all dynamic parameters in the model. The first element of each pair is the parameter name; the second element is a description of the parameter.

3.9 Event Bus API

By default, ADL models send and receive event bus data to all other registered cores. Each core that is instantiated is automatically registered and will receive data on a **send** operation. For platform integration, however, it may be necessary for the external model to handle the distribution of this data. In order to create code for the following API, the model must be generated with the **--no-event-bus-send** option.

Event bus data may be sent by the external environment to a core by calling the following method:

```
<core-class>::event_bus_send(const <event-bus-type> &t);
```

where **<core-class>** is the name of the core's class and **<event-bus-type>** has the form **<event-bus-name>_t**. This will invoke the core's handler for this event bus.

In order to access the event bus data types, you must include the generated ADL model as a header. If you are also compiling the model separately, then define `ADL_HEADER` before including the file in order to avoid duplicate definitions. For example, given a model with a core class name of **Core** and an event bus named **foobar** which has a field named **addr**:

> #define ADL_HEADER #include "model.cc"
>
> ... Core::foobar_t x; x.addr = ... ; core->event_bus_send(x); ...

With the **--no-event-bus-send** option set, data will be sent to a registered functor object and not automatically to other cores. The external code must also register an object derived from **<core-class>::EventBusReceiver**. This object has **operator()** overloaded for each type of event bus, e.g.:

```
struct EventBusReceiver {
  void operator()(<core-class> &core,const foobar_t &data) {};
};
```

On a send operation performed by the core, the appropriate method in **EventBusReceiver** will be invoked. To register this object, call **<core-class>::setEventBusReceiver**.

3.10 External Interrupt Generation

Exceptions may be generated by an external application. The function to do this is `IssNode::genExceptions(unsigned flag)`. The argument specifies what interrupts should be taken as a set of bit flags. These flags correspond to the `_flag` values listed in the introspection information which may be obtained by calling `getExceptionInfo()`.

When an exception is generated, the side-effects of the interrupt are immediately performed if the exception is enabled, as specified by its enable predicate, if present. The next time the model is cycled, these changes will be taken into account. However, if called during the execution of an instruction, such as within a call to a `MemHandler` method, this does not abort the execution of an instruction. To do this, call `IssNode::genExceptionImm(unsigned id)`. This generates the specified exception, aborting the instruction currently being executed. This should only be called if in the middle of an instruction execution.

If an exception is disabled and is specified as being level sensitive, then the side-effects of the exception will not immediately occur. Instead, the exception will remain pending until the exception is enabled or it is canceled. To cancel an exception, call `IssNode::cancelException(unsigned flag)`. As with `genException`, the argument is considered a set of bit flags. in this case identifying the exceptions to be canceled.

3.11 External Model Communication API

An `unsigned int` flag is provided as a means of communication between the ISS and an external application, such as a performance model. Within the ISS, action code may set the flag by calling `setExtInstrFlag(unsigned)` and retrieve it by calling `getExtInstrFlag()`.

When a transactional or hybrid ISS is generated, these functions are declared virtual so that a class derived from the ISS may implement them. This can be useful, for

example, to allow the ISS to communicate MMU translation status or other types of state to an instruction object within a pipeline model.

4 Internal and External Memories

4.1 External Memory Interface

By default, an ADL ISS contains a memory model with support for 64-bit addresses. A model may also be generated such that any or all memories are defined externally. This may be accomplished by either using the `--extern-mem` option with `adl2iss` or by setting the `extern_mem` flag to true for a memory within an ADL ISS configuration file. For more information about configuration files, please refer to the ADL ISS manual.

If a memory is defined externally then the external application must derive a class from the `MemHandler` interface and supply it to each core object by calling the `setMemHandler` method. This interface class is declared in `MemHandler.h` and consists of the following methods:

- *uint64 read64(CacheAccess ca,addr_t ra)*: Read a 64-bit value from memory.
- *uint32 read32(CacheAccess ca,addr_t ra)*: Read a 32-bit value from memory.
- *uint16 read16(CacheAccess ca,addr_t ra)*: Read a 16-bit value from memory. Note that this address may be misaligned within a word.
- *uint8 read8(CacheAccess ca,addr_t ra)*: Read an 8-bit value from memory.
- *void write64(addr_t ra, uint64 v, uint64 m)*: Write a 64-bit value to memory. The mask `m` specifies which bits of the 64-bit value `v` should be written.
- *void write32(addr_t ra, uint32 v, uint32 m)*: Write a 32-bit value to memory. The mask `m` specifies which bits of the 32-bit value `v` should be written.
- *void write16(addr_t ra, uint16 v, uint16 m)*: Write a 16-bit value to memory. The mask `m` specifies which bits of the 16-bit value `v` should be written. Note that this address may be misaligned within a word.
- *void write8 (addr_t ra, uint8 v)*: Write an 8-bit value to memory.
- *void readpage(CacheAccess ca,byte_t *t,size_t n,addr_t addr,addr_t crit_addr)*: Read a block from memory into the target `t` of size `n` bytes, starting at address `addr`, where `crit_addr` is the critical address which generated the request.
- *void writepage(byte_t *s,size_t n,addr_t addr,addr_t crit_addr)*: Write a block from source `s` into external memory of size `n` bytes, at address `addr`, where `crit_addr` is the critical address which generated the request.
- *void reset()*: Reset memory.
- *void set_latency(unsigned l)*: Set a latency value, if using this interface in conjunction with a performance model. The performance model will use the latency value that is set by this call when updating its internal time.

The read routines take an extra parameter of `CacheAccess` to specify whether the access is for instruction or for data. Currently, all instruction accesses are done using 32-bit accesses, so only calls of `read32` will ever indicate an instruction access. If the access is for an instruction fetch then a value of `CacheIFetch` will be supplied. For data, the value will be `CacheRead`.

4.2 Direct-Memory-Interface API

ADL provides a direct-memory-interface (DMI) cache for improving performance. This is an object which may cache memory accesses so that it is not necessary for the integration code to generate a memory transaction for simple memory accesses.

This feature is implemented when a model is generated with the `--extern-dmi` command-line option or `extern_dmi=<bool>` configuration-file parameter. The entry-size of the cache may be set via `--extern-dmi-entry-size=<int>` (command-line) or `extern_dmi_entry_size=<int>` (configuration file). It is expressed in terms of log(2) bytes, e.g. a value of 7 implies an entry size of 128 bytes. The total number of entries may be set via `--extern-dmi-size=<int>` (command-line) or `extern_dmi_size=<int>` (configuration file). Internally, a separate cache is used for data reads, data writes, and instruction data.

The size of the DMI cache may be overridden at build-time by using a preprocessor define. Thus, if a model is distributed as source, the final user of the model may modify the entry-size and number of entries without rebuilding the model. Simply redefine **DMI_HANDLER_TYPE**. This must be done consistently if the model is used as both a header and as a source file. The definition has the form `DmiMemHandler<DmiMemType,DmiEntrySize,DmiSize>`, where the first template parameter (DmiMemType) may be either **DmiInternalMem** or **DmiExternalMem**, based upon whether an internal or external memory is being used, and `DmiEntrySize` and `DmiSize` represent the entry-size and number of entires in the DMI cache, and exist as enums defined within the model's class.

The DMI cache may be used with internal or external memory, but only really makes sense with an external memory interface, such as for a platform integration. The internal-memory support is primarily meant for testing; it allows the user to build a standalone executable in order to check that the DMI cache is functioning properly.

On a memory access (one which misses in a hardware cache, if there is one) the appropriate DMI cache is checked. If the data is not contained in the DMI cache, the appropriate miss-handler routine is invoked. For an external memory, this means that the appropriate method of `MemHandler` is called. During this call, the DMI cache may be updated by calling `IssNode::update_dmi_cache_entry(CacheAccess ca,addr_t addr,byte_t *mem)`. The `CacheAccess` parameter should be the same as what was supplied to the `MemHandler` method, so that the correct DMI cache is updated.

The `addr` parameter and `mem` pointer should correspond, i.e. the pointer should point to the exact host memory location containing the target's data at location `addr`. The memory data should be in host endian format and the memory block should be at least as large as the DMI entry size.

Note that, for a model which is built with an internal DMI software cache which maps effective (virtual) addresses directly to memory, and if that cache interfaces directly with memory, i.e. there is no hardware data cache, the update function will also update this cache. Therefore, any calls to the update function should only be done during a `MemHandler` invocation so that the core will be able to associate an effective address with the DMI update.

The basic idea is that the cache is populated as misses occur. The user's `MemHandler` class should only populate the cache with "simple" memory, i.e. memory which does not map to peripherals or other memory-mapped registers. If the system performs access checks on memory transactions, then those access checks will be performed by the initial miss. After that, the access is known to be safe and thus the entry may be placed in the cache. However, if the system's configuration changes such that access permissions are modified, then the DMI cache must be updated. It may either be completely reset or specific entries may be invalidated via `IssNode::invalidate_dmi_cache_entry(CacheAccess ca,addr_t addr)` or `IssNode::invalidate_dmi_cache()`. For a model which contains an internal DMI cache which interfaces directly with memory (i.e. there is no hardware data cache), both invalidate functions will reset the internal cache, since the effective address for the invalidate is not known.

5 System-Call Support

ADL models may contains a mechanism for implementing system-calls. This is done via a simple memory-mapped interface: If a write occurs to the designated *porthole* address, then the function `handle_porthole()` is called. It is up to the external application to query memory in order to read the arguments of the system call.

By default, system-call support is enabled but turned off. It may be activated by setting the global variable `Porthole_enabled` to true and setting the global variable `Porthole_address` to the desired address value.

6 File Readers

The various file readers (DAT, UVP, and ELF) supported by ADL are normally not linked in with a bare-bones ISS. The **adl-config** option **--readers** may be used to specify these extra libraries. This option is only relevant with the **--basic** option, as otherwise the libraries are included by default.

For example, to use ADL's ELF reader within an external program:

```
#include "iss/ElfReader.h"

...

IssNode *root = IssNamespace::createTopLevelNode(id);

...

ElfReaderFactory erf;
Reader *er = erf.create(filename,*root);
er->useFileEntryPoint(true);
if (!er->readFile()) {
  throw runtime_error("Could not read ELF file '" + filename + "'.");
}
```

To obtain the C flags for the compilation process:

```
adl-config --cflags --basic
```

For linking:

```
adl-config --libs --basic --readers
```

7 A Simple Example

The following is an example which demonstrates the use of the ADL ISS API. The program instantiates a model, initializes the external memory and the model, runs the simulation, then displays various results.

```
//
// Copyright (C) 2005 by Freescale Semiconductor Inc.  All rights reserved.
//
// You may distribute under the terms of the Artistic License, as specified in
// the COPYING file.
//

//
// This is a simple driver program that can link against a bare-bones ISS
// that does not have the standalone infrastructure.
//
// You may define IssNamespace to specify an alternate namespace to be included.
//

#include <assert.h>
#include <iostream>
#include <iomanip>
#include <stdarg.h>
#include <sstream>
#include <stdexcept>
#include <memory>

//#define VLE

#ifdef USE_LOGGING_MGR
# include "rnumber/RNumber.h"
#else
```

```
# define __NO_RNUMBER__
#endif

#include "helpers/Macros.h"

#ifndef IssNamespace
# define IssNamespace DEFAULT_ISS_NAMESPACE
#endif

#define ISS_NAMESPACE IssNamespace
#include "iss/ModelInterface.h"
#include "iss/Memory.h"
#include "iss/MemHandler.h"

#ifdef UADL
# include "uadl/UadlArchIf.h"
#endif

using namespace std;
using namespace adl;

//
// For this example, the memory is implemented as a 1 Mb array.  The byte_*
// routines are helper functions declared in Memory.h.
//

// External memory.
enum { MemSize = 0x100000, MemMask = (MemSize-1) };

bool MemLogging = true;

byte_t external_mem[MemSize];

uint32_t extern_mem_read32(addr_t ra)
{
  return byte_read32(external_mem,(ra & MemMask));
}

void extern_mem_write32(addr_t ra, uint32_t v, unsigned nb = 4)
{
  byte_write32(external_mem,(ra & MemMask),v,nb);
}

struct MyMem : public MemHandler {

  MyMem(IssNode *node,unsigned excpt,addr_t exception_addr = 0) : _node(node), _excpt(excpt), _exception

  uint64_t read64(CacheAccess ca,addr_t ra)
  {
    log_access(ca,ra & ~DWMask,64);
    return byte_read64(external_mem,(ra & MemMask));
  }

  uint32_t read32(CacheAccess ca,addr_t ra)
  {
    log_access(ca,ra & ~WordMask,32);
    if (_excpt && ra == _exception_addr) {
      _node->genExceptionImm(_excpt);
    }
    return byte_read32(external_mem,(ra & MemMask));
  }

  uint16_t read16(CacheAccess ca,addr_t ra)
  {
```

```
  log_access(ca,ra & ~HWMask,16);
  return byte_read16(external_mem,(ra & MemMask));
}

uint8_t  read8(CacheAccess ca,addr_t ra)
{
  log_access(ca,ra,8);
  return byte_read8(external_mem,(ra & MemMask));
}

void write64(addr_t ra, uint64_t v, unsigned nb)
{
  log_access(CacheWrite,ra & ~DWMask,64);
  byte_write64(external_mem,(ra & MemMask),v,nb);
}

void write32(addr_t ra, uint32_t v, unsigned nb)
{
  log_access(CacheWrite,ra & ~WordMask,32);
  byte_write32(external_mem,(ra & MemMask),v,nb);
}

void write16(addr_t ra, uint16_t v, unsigned nb)
{
  log_access(CacheWrite,ra & ~HWMask,16);
  byte_write16(external_mem,(ra & MemMask),v,nb);
}

void write8 (addr_t ra, uint8_t  v)
{
  log_access(CacheWrite,ra,8);
  byte_write8(external_mem,(ra & MemMask),v);
}

void readpage(CacheAccess ca,byte_t *t,size_t n,addr_t addr,addr_t crit_addr)
{
  log_access(ca,addr,crit_addr,n);
  byte_pagecopy(t,0,external_mem,(addr & MemMask),n);
}

void writepage(byte_t *s,size_t n,addr_t addr,addr_t crit_addr)
{
  log_access(CacheWrite,addr,crit_addr,n);
  byte_pagecopy(external_mem,(addr & MemMask),s,0,n);
}

void reset()
{
  memset(external_mem,0,MemSize);
}

void log_access(CacheAccess ca,addr_t ra,int size)
{
  if (MemLogging) {
    switch (ca) {
    case CacheIFetch:
      cout << "# Instruction read:  0x" << hex << ra << "\n";
      break;
    case CacheWrite:
      cout << "# Data write (" << dec << size << " bits):  0x" << hex << ra << "\n";
      break;
    case CacheRead:
      cout << "# Data read (" << dec << size << " bits):  0x" << hex << ra << "\n";
      break;
```

```
      case CacheLogRead:
        cout << "# Data log read (" << dec << size << " bits):  0x" << hex << ra << "\n";
        break;
      case CacheILogRead:
        cout << "# Instruction log read:  0x" << hex << ra << "\n";
        break;
      default:
        ;
      }
    }
  }

  void log_access(CacheAccess ca,addr_t ra,addr_t crit_ra,int size)
  {
    if (MemLogging) {
      switch (ca) {
      case CacheIFetch:
        cout << "# Instruction read:  0x" << hex << ra << ", critical:  0x" << crit_ra << "\n";
        break;
      case CacheWrite:
        cout << "# Data write (" << dec << size << " bits):  0x" << hex << ra << ", critical:  0x" << cr
        break;
      case CacheRead:
        cout << "# Data read (" << dec << size << " bits):  0x" << hex << ra << ", critical:  0x" << cri
        break;
      case CacheLogRead:
        cout << "# Data log read (" << dec << size << " bits):  0x" << hex << ra << ", critical:  0x" <<
        break;
      case CacheILogRead:
        cout << "# Instruction log read:  0x" << hex << ra << ", critical:  0x" << crit_ra << "\n";
        break;
      default:
        ;
      }
    }
  }

private:
  IssNode *_node;
  unsigned _excpt;
  addr_t   _exception_addr;

};

#ifdef USE_LOGGING_MGR

#include "iss/LoggingMgr.h"

struct MyLogger : public LogParentIface, public LoggingIface {

  addr_t _instr_ea;

  LoggingIface *register_core(IssCore &core)
  {
    return this;
  }

  void logInstrPrefetch(addr_t ea)
  {
    cout << "\nI ea=0x" << hex << ea << "\n";
  }

  void logInstrRead(unsigned id,const char *name,addr_t ea,
                    addr_t ra,uint32_t data)
```

```
  {
    _instr_ea = ea;
  }

  void logInstr(const uint32_t* opc,int num_half_bytes,const char *name,Disassembler dis)
  {
    cout << "INSTR op=0x" << hex << setfill('0') << setw(num_half_bytes) << (*opc >> ((8-num_half_bytes)
    dis(cout,_instr_ea,opc);
    cout << "\"\n";
  }

};

#else

//
// These are the logging functions.  For this example, they simply display their
// results in a format similar to the DAT trace format used by the standalone ADL
// ISS executables.
//

static addr_t instr_ea, instr_ra;

namespace adl {

struct MyLogger : public LogBase {

  void log_instr_prefetch(addr_t ea)
  {
    cout << "\nI ea=0x" << hex << ea << "\n";
  }

  void log_instr_read(unsigned id,const char *name,addr_t ea,
                      addr_t ra,uint32_t data)
  {
    instr_ea = ea;
    instr_ra = ra;
    cout << "M n=Mem t=ifetch ea=0x" << hex << ea << " ra=0x"
         << ra << " d=0x" << data << "\n";
  }

  void log_instr(const uint32_t* opc,int num_half_bytes,const char *name,Disassembler dis,uint32_t flags
  {
    cout << "INSTR op=0x" << hex << setfill('0') << setw(num_half_bytes) << (*opc >> ((8-num_half_bytes)
    dis(cout,instr_ea,opc);
    cout << "\"\n";
  }

  void log_reg_write(unsigned id,const char *name,uint64_t value)
  {
    cout << "R n=" << name << " d=0x" << hex << value << "\n";
  }

  void log_regfile_write(unsigned id,const char *name,uint32_t index,
                         uint64_t value)
  {
    cout << "R n=" << name << " i=" << dec << index
         << " d=0x" << hex << value << "\n";
  }

  void log_core_mem_write(unsigned id,const char *name,
                          addr_t ea,int nb)
  {
    cout << "D n=Mem t=write ea=0x" << hex << ea
```

```
       << " nb=" << dec << nb << "\n";
}

void log_core_mem_read(unsigned id,const char *name,
                       addr_t ea,int nb)
{
  cout << "D n=Mem t=read ea=0x" << hex << ea
       << " nb=" << dec << nb << "\n";
}

void log_mem_write(unsigned id,const char *name,bool pre,int seq,addr_t ea,
                   addr_t ra,uint32_t value)
{
  if (!pre) {
    cout << "M n=Mem t=write ea=0x" << hex << ea
         << " ra=0x" << ra << " d=0x" << value << "\n";
  }
}

void log_mem_read(unsigned id,const char *name,bool pre,int seq,addr_t ea,
                  addr_t ra,uint32_t value)
{
  if (!pre) {
    cout << "M n=Mem t=read ea=0x" << hex << ea
         << " ra=0x" << ra << " d=0x" << value << "\n";
  }
}

void log_annotation(MsgType type,unsigned level,const std::string &msg)
{
  cout << "A l=" << level << " m=\"" << msg << "\"\n";
}

void log_cache_action(const char *name,CacheAction action,CacheAccess access,unsigned level,
                      int set,int way,unsigned linemask,addr_t ra)
{
  cout << "C n=" << name;
  switch (action) {
  case CacheLoad:
    cout << " a=load";
    break;
  case CacheHit:
    cout << " a=hit";
    break;
  case CacheMiss:
    cout << " a=miss";
    break;
  case CacheEvict:
    cout << " a=evict";
    break;
  case CacheNone:
    break;
  }

  switch (access) {
  case CacheNoAccess:
    cout << " t=none";
    break;
  case CacheIFetch:
    cout << " t=ifetch";
    break;
  case CacheRead:
    cout << " t=read";
    break;
```

```
    case CacheStore:
    case CacheWrite:
      cout << " t=write";
      break;
    case CacheFlush:
      cout << " t=flush";
      break;
    case CacheTouch:
      cout << " t=touch";
      break;
    case CacheAlloc:
      cout << " t=alloc";
      break;
    case CacheInvalidate:
      cout << " t=invalidate";
      break;
    case CacheLock:
      cout << " t=lock";
      break;
    case CacheUnlock:
      cout << " t=unlock";
      break;
    default:
      break;
    }

    cout << dec;

    if (set >= 0) {
      cout << " set=" << set;
    }
    if (way >= 0) {
      cout << " way=" << way;
    }

    cout << " lm=0x" << hex << linemask << " ra=0x" << (ra & ~((addr_t)linemask)) << dec << "\n";
  }

};

}

#endif

MyLogger mylogger;

//
// These are some helper functions for displaying model state.
//

void display_memory(int cnt, ... )
{
  va_list ap;
  va_start(ap,cnt);
  for (int i = 0; i != cnt; ++i) {
    unsigned addr = va_arg(ap,unsigned);
    cout << "MD n=Mem ra=0x" << hex << addr << " d=0x" << hex
         << extern_mem_read32(addr) << "\n";
  }
  va_end(ap);
}

struct Reporter : ReportBase
{
```

```
    virtual  void report_reg (const IssNode *,unsigned id,const char *name,REGTYPE value,bool shared) {
      cout << "RD n=" << name << " d=0x" << hex << setfill('0') << setw(8) << value << "\n";
    }

    virtual  void report_regfile (const IssNode *,unsigned id,const char *name,int index,REGTYPE value,b
      cout << "RD n=" << name << " i=" << dec << index << " d=0x" << hex << setfill('0') << setw(8) << v
    }

    virtual void report_cache(const IssNode *,const char *name,CacheType type,unsigned level,addr_t addr
                              bool valid,const FieldData &fd,bool shared,const byte_t *data,unsigned n)
      if (valid) {
        cout << "CD n=" << name << " set=" << dec << set << " way=" << way << hex
             << " ra=0x" << setfill('0') << setw(sizeof(addr_t)*2) << setfill('0') << addr << dec;
        ForEach(fd,i) {
          cout << " " << i->_name << "=" << i->_value;
        }
        cout << " d=" << hex;
        bool first = true;
        for (unsigned i = 0; i < n; i += 4) {
          if (!first) cout << ",";
          first = false;
          cout << "0x" << setw(8) << setfill('0') << byte_read32(data,i);
        }
        cout << dec << "\n";
      }
    }
};

unsigned get_exception_id(IssNode *node,const string &name)
{
  const ExceptionInfos &einfo = node->getExceptionInfo();
  ForEach(einfo,i) {
    if (i->_name == name) {
      return i->_id;
    }
  }
  RError("Unknown exception name " << name);
}

int main(int argc,const char *argv[])
{
  try {
  bool trace = true;
  int prog = 0;
  unsigned id = 0;

  for (int i = 1; i != argc; ++i) {
    if (!strcmp(argv[i],"-trace")) {
      trace = true;
    }
    else if (!strcmp(argv[i],"-no-trace")) {
      trace = false;
    }
    if (!strcmp(argv[i],"-mem-logging")) {
      MemLogging = true;
    }
    else if (!strcmp(argv[i],"-no-mem-logging")) {
      MemLogging = false;
    }
    else if (!strcmp(argv[i],"-prog")) {
      prog = atoi(argv[++i]);
    }
  }
```

```
  // This instantiates the model.
# ifdef UADL
  uadl::UadlArch *model = uadl::createArch("P");
  IssNode *root = &model->ISS();
# else
  IssNode *root = IssNamespace::createTopLevelNode(id);
# endif


# ifdef TrapAddr
  MyMem mem(root,get_exception_id(root,"DataStorage"),TrapAddr);
# else
  MyMem mem(root,0,0);
# endif

# ifdef USE_LOGGING_MGR
  auto_ptr<LoggingMgr> mgr(createLoggingMgr(false));
  mgr->installLoggers(root);
# else
  root->setLogger(&mylogger);
# endif

  root->setMemHandler(&mem);

  if (trace) {
    root->set_tracing_r();
  }

  cout << "CORE n=:P\n";

  // This initializes memory to the program to be run.
# ifdef VLE
  extern_mem_write32(0x00000100,0x04897f5b);
  extern_mem_write32(0x10000104,0xd051e2fd);

  root->setReg("NIA",0,0x100);
  root->setReg("GPR24",0,30);
  root->setReg("GPR25",0,100);
  root->setReg("GPR26",0,10);
  root->setReg("GPR27",0,1);
# else

  switch (prog) {
  case 0:
    extern_mem_write32(0x00000100,0x80401000);  // lwz r2,0x1000(0)
    extern_mem_write32(0x00000104,0x394A0100);  // addi r10,r10,0x100
    extern_mem_write32(0x00000108,0x7C620A14);  // add r3,r2,r1
    extern_mem_write32(0x0000010c,0x34A5FFFE);  // subic. r5,r5,2
    extern_mem_write32(0x00000110,0x90640000);  // stw r3,0(r4)
    extern_mem_write32(0x00000114,0x90660000);  // stw r3,0(r6)
    extern_mem_write32(0x00000118,0xA0E01004);  // lhz  r7,0x1004(r0)
    extern_mem_write32(0x0000011c,0x89001005);  // lbz  r8,0x1005(r0)
    extern_mem_write32(0x00000120,0x99002004);  // stb  r8,0x2004(r0)
    extern_mem_write32(0x00000124,0xB0E02008);  // sth  r7,0x2008(r0)
    // Some register initializations.
    root->setReg("NIA",0,0x100);
    root->setReg("GPR1",0,0x1fff);
    root->setReg("GPR4",0,0x2000);
    root->setReg("GPR6",0,0xfffffffe);
    // If there's a cache, enable it.
    root->setReg("CCR",0,0xc0000000);
    break;
  case 1:
    extern_mem_write32(0x00000100,0x80201000);  // lwz r1,0x1000(r0)
```

```
    extern_mem_write32(0x00000104,0x80403000);  // lwz r2,0x3000(r0)
    extern_mem_write32(0x00000108,0x80601004);  // lwz r3,0x1004(r0)
    extern_mem_write32(0x0000010c,0x00000000);  // .long 0
    extern_mem_write32(0x00000110,0x00000000);  // .long 0
    extern_mem_write32(0x00000114,0x7D5A02A6);  // mfspr r10,SRR0
    extern_mem_write32(0x00000118,0x394A0004);  // addi r10,r10,4
    extern_mem_write32(0x0000011c,0x3A801234);  // li r20,0x1234
    extern_mem_write32(0x00000120,0x7D5A03A6);  // mtspr SRR0,r10
    extern_mem_write32(0x00000124,0x4C000064);  // rfi
    // Some register initializations.
    root->setReg("NIA",0,0x100);
    root->setReg("IVOR2",0,0x114);
    break;
  case 2:
    extern_mem_write32(0x00000010,0x8040100C); // lwz r2,0x100c(r0)
    extern_mem_write32(0x00000014,0x8040200C);  //  lwz r2,0x200c(r0)
    extern_mem_write32(0x00000018,0x90403008); // stw r2,0x3008(r0)
    root->setReg("NIA",0,0x10);
    root->setReg("CCR",0,0xc0000000);
    break;
  default:
    RError("No program " << prog << " defined.");
  }

# endif

  // Data memory initializations.
  extern_mem_write32(0x1000,0xdeadbeef);
  extern_mem_write32(0x1004,0x12345678);
  extern_mem_write32(0x2000,0x00000000);
  extern_mem_write32(0x3000,0x87654321);
  extern_mem_write32(0xfffffffc,0);
  extern_mem_write32(0x00000000,0);

  cout << "TRACE\n";

# ifdef USE_LOGGING_MGR
  // If using the logging manager, then install our logger.
  mgr->registerLogger(&mylogger);
# endif

  // This runs the simulation until it halts.  For this model, the core will
  // halt when it encounters an opcode of 0x00000000.
# ifdef UADL
  while (model->isActive()) {
    model->proceed();
  }
# else
  root->run();
# endif

  cout << "\nRESULTS\n";

# ifdef UADL
  cout << "\n# time " << model->currTime() << ": simulation terminated.  "
       << model->issueCount() << " instructions issued.\n\n";
# endif

  // Final results display.
  Reporter reporter;

  display_memory(6,0x1000,0x2000,0x2004,0x2008,0xfffffffc,0x00000000);
  root->showRegs(reporter);
  root->showCaches(reporter);
```

```
  }
  catch (exception &err) {
    cerr << "Error:  " << err.what() << "\n\n";
    return 1;
  }
}
```

The input model is a stripped-down version of a PowerPC. The model library is created using the command:

```
adl2iss mod1.adl --output=mod1-base.so --target=base-so --no-debug-mode
--no-rnumber --extern-mem --no-porthole-enabled
```

which results in a file named `mod1-base.so`. The driver program can be compiled in the following manner (assuming that the input file is named `driver.C`):

```
g++ `adl-config --cflags` driver.C -c -o driver.o
```

The final linking is done in the following manner:

```
g++ -o driver driver.o mod1-base.so
```

Generated on: 2018/02/28 15:17:11 MST.