The ADL Trace Format

A Description of DAT Files

This document describes the ADL trace file format, known as DAT. This is a simple textual format capable of describing a microprocessor system's initial state, events which occur during simulation, and a final result state. The format is designed to be easily parseable by common scripting languages such as Perl and to be extensible so that as new resource types are added to ADL, the DAT format can be extended to handle them.

Author: Brian Kahne
Contact: bkahne@freescale.com

Table of Contents

1 The File Format

1.1 Overview

The DAT format is a simple, line-oriented textual format for describing the initial and final state of all resources within a microprocessor system, as well as a stream of instructions which occurs during a simulation and the modifications of resources which occur as each instruction executes. The format is stream oriented, meaning that a simulation does not have to store large data structures in memory in order to

generate the file, thus allowing it to scale to long simulation runs. Although the fact that it is human-readable does mean that it is less compact than a binary format, it can be coupled with a compression library, such as zlib, in order to reduce file sizes.

The basic unit of data within the file is a line. Each line may start with a unique numerical identifier, used to group together resource changes. It is then followed by a command and then data relevant to a command. The format is case-sensitive and no line continuation features exist. The radix of integer constants follows the C convention of "0x" for base-16, no prefix for base-10, etc.

A parser for the DAT format should be able to operate in a forward-compatible manner. This means that unknown command types should be ignored and should not cause an error so that if the format is extended in the future, older parsers will still be able to process these newer files. It is permissible to display warning messages about unknown command types and to have a mode in which unknown command types cause errors, but the default mode of operation should be to ignore the commands.

Other notes on syntax:

- All addresses may be 64-bit values.
- All memory accesses are 32-bit values.
- Register data is specified as a single number of the size appropriate for the register. This means that a 128-bit register is specified as a single 128-bit value.
- All strings are quote-delimited, e.g. "lwz 1,2,3".

1.2 Comments

Single-line comments start with "#" and may occur on their own or on the same line after a non-comment. They should always be ignored by a parser.

Multi-line comments start with "=", followed by a unique identifier tag. The comment ends with "= /<tag>". Nested tags are allowed, but they should end in the reverse order in which they were begun.

An example single-line comment:

```
#
# This is a comment.
#
```

An example multi-line comment:

```
= foo

  This is all
  comment data
  here.
```

```
= /foo
```

While single-line comments are meant to be comments, i.e. they should always be ignored by a parser, the intent of multi-line comments is to provide a facility for embedding non-trace data into files for processing by other programs. For example, an *asm* tag might be used to denote a program in assembly language. A program might extract out all data within the *asm* blocks, feed this to an assembler, and then place the results back into the DAT file.

1.3 Commands

A command takes the form of:

```
[id.] <command> <data>
```

Where *id* is a unique integer, *command* is an identifier, and *data* consists of key-value pairs of information. The order of the key/value pairs is not important. A key-value pair consists of a key that is a legal C identifier (must start with an alphabetical character or "_" but then may contain digits after that), followed by "=" and then a value. No space is allowed on either side of the "=". The value may be a valid identifier, a number, or a double-quote delimited string.

Some example commands:

```
# An initialization of a memory named "Mem" at address 0x1000,
# of the value 0xdeadbeef.
MD n=Mem ra=0x1000 d=0xdeadbeef

# A write to "Mem" at 0x1000 of the value 0x1234567
1. D n=Mem t=Write ra=0x1000 d=0x12345678
1. A l=1 m="ThreadSwitch Reason: L2miss"
```

1.4 Sections

A DAT file is structured into sections. The outer-most section type, a test, encompasses a complete simulation run. Within a test is an initial state, the trace, and the final result state. Sections may be parsed using a simple finite-state-machine approach: There is no need to use a stack to keep track of section information. However, keep in mind that it is valid for initial, trace, and result sections to be interleaved. In other words, there may be multiple initial, result, and trace sections and *CORE* directives, which specify the active core, may also be interleaved.

2 Recognized Commands

This section lists all commands currently recognized by the format.

2.1 Section Commands

- **TEST**: Specifies that a new test has begun. A leading TEST command in a file is optional.
  - **id=<id>**: The test number.
- **INIT**: Specifies that subsequent information is for initialization purposes.
- **TRACE**: Specifies that subsequent information is for trace purposes.
- **RESULT**: Specifies that subsequent information is for specifying the final result state.
- **CORE**: Specifies that subsequent information applies to the specified component in the model.
  - **n=<name>**: A path identifying a model component. This takes the form of a colon-delimited string, e.g. "proc0:thread0".
- **CTX**: Specifies that subsequent information applies to the specified context of the current core.
  - **n=<name>**: The context's name.
  - **cn=<int>**: The active context number/index.
- **NOCTX**: Specifies that subsequent information is no longer context specific.

2.2 Initial/Final State Commands

- **MD**: Memory data.
  - **n=<name>**: The name of the memory. The default, global memory is named "Mem".
  - **ra=<addr>**: The real-address of the data.
  - **d=<data>**: The data, a 32-bit value.
  - **s=<int>**: Optional size specifier, in bytes. Allowed values are 1, 2, and 4. If omitted, default is 4.
- **CD**: Cache data. This specifies data for a cache line.
  - **n=<name>**: The cache name, e.g. L2.
  - **set=<int>**: The set of the cache line.
  - **way=<int>**: The way of the cache line.
  - **ra=<addr>**: The real-address of the cache line.
  - **valid=<0|1>**: The valid status of the line.
  - **dirty=<0|1>**: The dirty status of the line.
  - **data=<word,word...>**: A sequence of 32-bit data words specifying the contents of the cache line.
- **TD**: TLB data. This specifies information about an MMU translation.
  - **n=<name>**: The name of the translation, e.g. TlbCam.
  - **set=<int>**: The set of the translation.
  - **way=<int>**: The way of the translation.
  - **<field>=<int>**: Additional fields are considered fields of the translation and are used to initialize the entry.

- **RD**: Register data.
  - **n=<name>**: The name of the register.
  - **i=<int>**: Index of the entry if the name specifies a register-file. This is optional: The index can be appended to the register-file name, if the file is non-sparse. In other words, if a register-file named `GPR` exists, then `n=GPR0` refers to the first element, as well as `n=GPR i=0`. For sparse register-files, this is not the case, since the individual elements of the file also exist as stand-alone registers.
  - **d=<data>**: The register data.

2.3 Trace Commands

2.3.1 External Register Write Events

- **ERW**: Start of an external register write, initiated by a call to `IssNode::writeReg`. What follows are intermediate results which occur as a result of the write operation. For example, a write-one-to-clear register might show a write value of 0xffffffff, but the intermediate results which follow will show that the resulting register now has a value of 0x00000000.
  - **d=<int>**: The updated data value.
  - **i=<int>**: The index of a register-file entry, if this is a register file.
  - **id=<int>**: A unique numerical identifier for this external-write event.

2.3.2 Instruction Events

- **I**: Start of an instruction sequence. This implies that all subsequent information, until a section command or another **I** or **ERW** command is encountered, is relevant to this instruction.
  - **id=<int>**: A unique numerical identifier for this instruction group.
  - **ea=<addr>**: The effective-address for the instruction fetch, if this instruction was fetched. Will be omitted if the instruction was executed via an external call (via `IssNode::exec_from_buffer`).
  - **tic=<int>**: A thread-local instruction count.
- **INSTR**: Information about an instruction.
  - **op=<int>**: The instruction's opcode.
  - **asm=<str>**: The disassembled instruction, e.g. "lwz r1,r2,r3".

2.3.3 Resource Update Events

- **A**: An annotation command. These are emitted by code within the model or by system calls generated by code running on the model.
  - **l=<int>**: The annotation level. This may be used to filter messages by priority.
  - **m=<str>**: The message string.

    - **t=<str>**: The type of message: `info`, `warning`, or `error`. Generally, if this is just an informational notice, then the type key is omitted. Also, if an error is generated, then the current behavior of the test writer is to abort the simulation at this point. Note that warnings and errors are always printed, even if tracing is turned off.
    - Additional keys may be present, specified by the user as data arguments in the *info*, *warning*, and *error* call.
- **B**: Branch information.
    - **ea=<addr>**: The branch target effective-address.
    - **taken=1**: Currently, this command is only issued when a branch is taken, thus `taken` will always be true. For future compatibility issues, though, we have left open the possibility that this will not be the case. Thus, an analysis program should always check the value of the `taken` parameter.
- **BP**: A breakpoint occurrence.
    - **ea=<addr>**: Effective address of the breakpoint.
- **C**: A cache access. Note that **C** commands precede in the file the memory access which causes them. Thus, a **C** command is tied to the next **INSTR** or **M** command in the file.
    - **n=<name>**: The name of the cache, e.g. L2.
    - **a=<hit|miss|evict>**: The action taking place.
    - **t=<read|write|flush|touch|alloc>**: The access type.
    - **set=<int>**: The cache line's set.
    - **way=<int>**: The cache line's way.
    - **lm=<int>**: The line mask for the cache.
    - **ra=<int>**: The real-address of the first word in the cache line.
- **D**: A memory data access. This describes a memory access from the point of view of the core and is then followed by **M** commands which represent the memory access from the point of view of the memory.
    - **n=<name>**: The name of the memory. The global memory is named `Mem`.
    - **ea=<addr>**: The effective-address of the access.
    - **nb=<int>**: The size of the access in bytes.
    - **t=<read|write>**: The access type.
    - **sn=<int>**: Optional. For a model which has dependency tracking enabled, this is the memory-read sequence number. Only present for reads.
- **E**: Specifies the occurrence of an exception.
    - **n=<name>**: The exception's name.
- **M**: A memory access.
    - **n=<name>**: The name of the memory. The global memory is named `Mem`.
    - **ea=<addr>**: The word-aligned effective-address of the access.
    - **ra=<addr>**: The word-aligned real-address of the access.
    - **t=<read|write|ifetch>**: The access type. Accesses of type `ifetch` occur between `I` and `INSTR` cards and represent the instruction-fetch operation. More than one `ifetch` **M** command may occur for misaligned instructions (if allowed by the architecture).

    - **d=<int>**: The data. This is a 32-bit word-aligned value representing the new state of memory.
- **R**: A register read or write. The syntax is the same as for the **RD** command in the init/result section, with the possible addition of an action.
    - **a=<read|write>**: The type of action: *read* or *write*. If omitted, then then the action is a write.
    - **d=<int>**: The updated data value.
    - **i=<int>**: The index of a register-file entry, if this is a register file.
    - **m=<int>**: A mask describing the portion of the register read. If omitted, then the entire register is accessed.
    - **sn=<int>**: Optional. For a model which has dependency tracking enabled, this is the register-read sequence number. Only present for reads.
    - **dt=<R|M>n[,<R|M>n...]**: Optional. For a model which has dependency tracking enabled, this lists the read dependencies. Register dependencies are expressed as **Rn**, where **n** is the register sequence number (listed as **sn=n** in the **R** logging event). Memory dependencies are expressed as **Mn**, where **n** is the memory sequence number (listed as **sn=n** in the **D** logging event).
- **T**: An MMU/TLB access. Note that **T** commands precede in the file the memory modification which causes them. Thus, a **T** command is tied to the next **INSTR** or **M** command in the file.
    - **n=<name>**: The name of the translation, e.g. TlbCam.
    - **<field>=<int>**: Additional fields are considered fields of the translation.
- **WP**: A watchpoint occurrence.
    - **ea=<addr>**: Effective address of the breakpoint.
    - **t=<read|write>**: Type of watchpoint.
    - **nb=<int>**: Number of bytes associated with the watchpoint memory access.
    - **d=<data>**: The value written, in the case of a write watchpoint.

Note that any resource update command may have an optional **cn** tag which will report the name of the core on which the resource modification is occurring, if it differs from the current core, as set by the **CORE** tag. This generally only occurs in multi-processor test-cases, where an event from one core causes modifications, e.g. cache activity, in other core within the execution of a single instruction.

A resource update may also have a **ctx** and **ctxi** tag if a resource in a context other than the current context is modified. In this case, **ctx** specifies the name of the context and **ctxi** specifies the context index.

2.4 Time-Tagged ISS and uAdL-Specific Trace Commands

These trace commands only occur within a time-tagged ISS.

- **ITIME**: For time-tagged ISSs, this is the time when an instruction is fetched.
    - **t=<time>**: Time of the fetch.

- **CTIME**: For time-tagged ISSs, this is the time when an instruction completes.
  - **t=<time>**: Time of the completion.

3 Test Generation

The ADL distribution contains a script called **makedat** designed to make it easy to create assembler-based tests. The basic idea is that one writes an assembly program within a multi-line comment block called *asm*. For example:

```
= asm

   addis 3,2,1
   addis 4,0,100
         mtspr 10,20
         mfspr 21,11
         mtspr 12,8

= /asm
```

Then, run the **makedat** script on this file:

```
makedat <input-file>
```

The script will default to using an assembler named **as** in the current directory. This way be overridden by using the **--as=<path>** option. By default, the script reads the specified input file, assembles the code found within the *asm* block, and then re-writes the file, adding the assembled code as **MD** commands to the file, surrounded by special comments. This allows the same input file to be updated repeatedly, while not overwriting anything else in the file. The user may then add in extra commands, such as register initializations, expected results, etc.

For example, here is a simple testcase with expected results:

```
#
# Example makedat testcase.
#
= asm

      addi r1,r1,10
      addi r2,r2,20
      add      r3,r1,r2
      add  r3,r3,r4

= /asm

# <GEN>
MD n=Mem ra=0x00000000 d=0x3821000A   #       addi r1,r1,10
MD n=Mem ra=0x00000004 d=0x38420014   #       addi r2,r2,20
MD n=Mem ra=0x00000008 d=0x7C611214   #       add      r3,r1,r2
MD n=Mem ra=0x0000000c d=0x7C632214   #       add  r3,r3,r4
# </GEN>

CORE n=:P

RD n=GPR i=4 d=40
```

```
RESULTS

RD n=GPR i=3 d=70
```

This shows the input assembly and the resulting assembled instructions. This also demonstrates initializing a register (GPR 4) to a value, and it shows an expected result (GPR3 is expected to be 70). Note the use of the **CORE** command: By default, **tracediff**, the DAT comparison tool, will associate results with the global scope unless a **CORE** directive is used to associate these results with a specific core.

The **makedat** script recognizes several other multi-line comment blocks:

- aopts: Place assembler flags within this block.
- mdopts: Place makedat command-line flags within this block. For example, to specify an offset for the assembled instructions:

  ```
  = mdopts
  instr-offset: 0x10000
  = /mdopts
  ```

4 Test Comparison

The ADL distribution contains a tool, **tracediff**, which can be used to compare DAT files. The usage is:

```
tracediff <expected-file> <simulation-output-file>
```

In other words, the first file is the testcase, such as the file shown in the prior section. The second file is the file produced by the simulator. **tracediff** will exit with an exit code of zero if the files compare, otherwise it will exit with a non-zero exit and will report where it found discrepancies.

If the simulation output file contains a trace section, then the expected-results file must contain a trace section. Otherwise, the trace section of the expected results will be ignored.

Generated on: 2018/02/28 15:14:27 MST.