Trace Protocol Specification
Version 2018-11-20
2e4f73bee02ededa076260146a8deb62156b9086

Darius Rad <darius@bluespec.com>

Thu Mar 7 15:11:07 2019 -0500

Preface

Warning! This draft is a work in progress.

Acknowledgments

I would like to thank Julie Schwartz for her valuable feedback.

The files used to generate this specification are heavily based on the RISC-V Debug Specification, by Tim Newsome.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is a draft specification for a RISC-V trace protocol that provides sufficient information for tandem verification and debugging. This is intended as an interim solution until the RISC-V foundation formally specifies adequate functionality.

This trace protocol encompasses complete instruction, data, and memory trace capabilities for a single hart. This protocol is also adaptable to implementations that cannot provide complete trace information (i.e., partial trace). While efficiency is important, it is not a primary goal of this protocol.

1.1 Observed State

One potential source of false positive tandem verification mismatch is related to the interaction between architectural state and a running program. To clarify this, a distinction is made between visible state and observed state.

Visible state is all architectural state. Visible state is all state that is potentially observable.

Observed state is state that directly affects a program running on a hart.

To illustrate the difference, consider the mip register which contains the interrupt pending bits. A hart is running in machine mode with mie bit in mstatus cleared. The meip bit of the mip register is initially clear. At some point later, the meip bit is set. As the hart has mie clear, it will not take an interrupt. Later, the meip bit becomes clear. Since the hart did not take an interrupt, as meip was clear, and did not read mip directly, the change in meip was never visible to the hart. The behavior of this hart would have been identical if, in the same scenario, meip was never set. Thus, this state change was never observed by the hart.

Visible states that may not be observed will commonly include interrupt pending registers, timers, and memory mapped peripheral registers.

Chapter 2

Trace Format

Opcode	Payload Length	Meaning
1	0	begin group (atomic state update)
2	0	end group
3	0	increment pc by instruction length (opcode 16, 17)
4	16 bits + (XLEN or FLEN)	full register (16 bit address, XLEN or FLEN data) (see notes)
5	24 bits	incremental register (16 bit address, 8 bit signed offset)
6	24 bits	incremental register (16 bit address, 8 bit OR mask)
7	varies	additional state (8 bit identifier, type dependent data length)
8	varies	memory request: addr (MLEN), op (4 bits), size (4 bits), data (0 for load, depends on size for store and amo)
9	16, 24, 40, or 72 bits	memory response: size (4 bits), result (4 bits), data (0 for store, depends on size for load and amo)
10	0	hart reset
11	0	state initialization
16	16 bits	instruction
17	32 bits	instruction

Memory Request Size	Meaning
0	8 bits
1	16 bits
2	32 bits
3	64 bits

Memory Request Opcode	Meaning
0	Load
1	Store
2	Load reserved (lr)
3	Store conditional (sc)
4	AMO swap
5	AMO add
6	AMO xor
7	AMO and
8	AMO or
9	AMO min
10	AMO max
11	AMO minu
12	AMO maxu
13	instruction fetch

Memory Request Result	Meaning
0	Success
1	Failure

Identifier	Data Length	Meaning
1	8 bits	privilege mode (PRIVLEN, zero extended)
2	32, 40, or 64 bits	physical address (MLEN, zero extended)
3	32, 40, or 64 bits	effective address (MLEN, zero extended)
4	8 bit	memory store data (see notes)
5	16 bit	memory store data
6	32 bit	memory store data
7	64 bit	memory store data
8	64 bit	mtime
9	32, 40, or 64 bits	pc physical address (MLEN, zero extended)
10	32 or 64 bits	program counter (pc)

Chapter 3

Notes

Trace data provides information for a single hart only.

XLEN, FLEN, and ILEN refer to the length in bits of an integer register, floating point register, and the maximum instruction length, as defined by the RISC-V user specification.

PRIVLEN is the length in bits to represent a privilege mode, which is 2.

MLEN is the length in bits necessary to represent any memory address. It is the maximum of the physical memory address length and virtual memory address length. MLEN width payload fields are rounded up to a multiple of 8 bits. MLEN for RV64 is 64. MLEN for RV32 is 34 when supervisor mode and Sv32 are supported, and 32 otherwise.

The trace encoder and decoder must agree on all parameters that impact trace data, including but not limited to XLEN, FLEN, ILEN, and MLEN. Negotiation of these parameters between the two is outside the scope of this protocol.

The tandem verifier (not the trace decoder) must be aware of elements that are absent from partial trace implementations. Negotiation of this information is outside the scope of this protocol.

All fields greater than one byte are transmit in little endian byte order. Fields less than one byte are packed with the first field starting at the least significant bit.

The trace protocol should be transferred on a reliable, in order transport mechanism. Flow control, checksumming, and retrying are outside the scope of this protocol.

State changes that occur between begin and end group occur atomically. State changes that occur outside begin and end group do not convey any ordering other than after the prior end group and before the next begin group. Only unambiguous state changes are allowed.

The end group opcode may be omitted if immediately followed by a begin group opcode. In other words, end group is implied by a begin group.

Unless otherwise noted, fields may be omitted, resulting in partial trace information. Unless otherwise noted, field ordering is not restricted.

The trace encoder is permitted but not required to use incremental formats.

Register addresses (opcodes 4, 5, 6) are as specified in the RISC-V Debug Specification. All GPR, FPR, and CSR registers are combined into a single 16 bit address space. The pc is represented as additional state.

Trace may include unobserved state (e.g., minstret, mtime, mip).

The instruction field represents the instruction that retired or caused an exception. It should be omitted for interrupts.

Memory response fields may be omitted. If present, a memory response field must immediately follow the corresponding memory request.

Increment PC (opcode 3) must be contained within a group that also contains an instruction field (opcode 16 or 17).

Memory trace information reflects implementation dependent behavior. As such, no assumptions can be made with respect to what transaction(s) a given instruction will generate. For example, AMOs may be implemented as read-modify-write and not as AMO transactions on the memory bus.

The additional state memory store data (opcode 7, indices 4, 5, 6, 7) is related to memory trace data, but the two represent conceptually different information. Memory store data refers to the instruction centric view of the data that would be written to memory in a given instruction. Memory trace data represents data from transactions on the system bus. The two may differ in certain circumstances, such as unaligned accesses or atomic memory operations, where the core may issue multiple bus transaction to accomplish a particular instruction. Ideal trace information, which would attempt to minimize inferences in the data measured, would observe memory store data in the core and would observe memory trace data on the system bus.

Certain implementations may make lazy updates to architectural state that appear out of order but do not impact program execution. For example, a slow divide operation may be retired by the processor, and subsequent instructions executed, before the divide operation is completed and the result is committed to the architectural state. This behavior may occur in out-of-order execution processors, but also may occur in in-order execution processors. The trace protocol defines that state updates occur in program order and thus requires that the trace encoder reorder state updates to match in order program execution. A hardware trace encoder may be assisted by a software portion that reorders such out of order state updates. However, only a trace that matches the format here, with state updates in program order, should be used as input to tandem verification or similar tools.

The state initialization opcode is used for changes to program state outside of the processor that must be copied to the shadow state but cannot be modeled. It is primarily intended for use during program loading by the debug module. The state initialization opcode should be within a group that includes the update, which would typically include a register opcode or a memory request and response opcode. The state initialization opcode must not be combined with opcodes related to program execution (i.e., instruction opcodes) or the reset opcode.

3.1 Tandem Verification

For complete tandem verification, all architectural side effects of an instruction or trap must always be included.

Memory trace information may be used to initialize memory in the tandem verifier.

Comparison of unobserved state must be done carefully to avoid false positive errors. Comparison of state should be done at the end of each group.

The recommended way to perform tandem verification using this trace protocol is by utilizing a shadow state in the tandem verifier. The verifier would apply all state changes, supplied by the trace decoder, to a copy of the entire architectural and microarchitectural state. For an implementation with full trace information, the shadow state would be identical to the state in the core generating the trace.

Some state elements within the architectural and microarchitectural state would be denoted as non-deterministic. These would typically be inputs related to hardware that is not modeled in the tandem verifier, such as timers and external interrupts. State classified as non-deterministic should be the minimum necessary to capture non-determinism, such as single registers or individual bits within registers.

Immediately prior to attempting to execute an instruction in the tandem verifier, the non-deterministic state should be copied from the shadow state to the tandem verifier state. This ensures that the tandem verifier will be able to match non-deterministic program flow changes, such as caused by interrupts, external inputs, or other behavior that is not modeled.

Tandem verification would operate as follows:

1. State changes are continually supplied by the trace decoder and are applied to the shadow state. This continues until all state changes within a begin/end group have been applied. Once the entire begin/end group state has been applied, continue to the next step.
2. Copy minimum non-deterministic state from the shadow state to the tandem verifier state.
3. Attempt to execute a single instruction in the tandem verifier.
4. Compare the complete shadow state and tandem verifier state.
5. If there is a mismatch between the shadow state and the tandem verifier state, report the difference. Then, copy the entire shadow state to the tandem verifier state.
6. Return to step 1.

State comparison may be done more efficiently by collecting state changes in both the shadow state and the tandem verifier state, and comparing these changes rather than the entire state. Note, however, that this must done in an instruction neutral way and must ensure that it will capture all state changes. In particular, the comparison must include erroneous state changes, and must not be limited to comparing the state elements that are expected to change.

3.2 Partial Trace

Partial trace may be supported by indicating some elements within the architectural and microarchitectural as absent. Changes to these state elements would not be present in the trace, and thus the shadow state would not accurately reflect those elements.

Tandem verification with partial trace would operate similar to that as described above, with the following changes:

1. State changes supplied by the trace decoder would not include elements that are absent.
2. Comparison of shadow state and tandem verifier state would exclude the elements that are absent.
3. In the event of mismatch, copying state from the shadow state to the tandem verifier state would exclude elements that are absent.

With partial trace it may not be possible to recover and continue tandem verification in all cases.

If the trace excludes state elements that are necessary to reflect non-deterministic behavior, then false positive tandem verification mismatches will result.

3.3 Future Considerations

The format could be optimized for efficiency.

The memory trace format could be enhanced to split memory request address and data, and memory response status and data. This could allow partial memory trace (address without data). It could also allow more efficient trace of large blocks of data (using an implied memory address).

Incorporate a mechanism for the encoder to convey all trace parameters (XLEN, FLEN, etc.) to the decoder.

Support multiple harts, perhaps by adding a hart identification opcode.

3.4 Tandem Verifier Interface

The tandem verification process will need to perform the following operations on the tandem verifier:

1. Read tandem verifier state, including registers and memory,
2. Write tandem verifier state, including resisters and memory,
3. Attempt to execute one instruction, and
4. Reset the tandem verifier.

Appendix A

Tandem Verifier Interface

The tandem verification process will need to perform the following operations on the tandem verifier:

1. Read tandem verifier state, including registers and memory,
2. Write tandem verifier state, including resisters and memory,
3. Attempt to execute one instruction, and
4. Reset the tandem verifier.

Appendix B

Trace Encoder Interface

The following example BSV code provides a suggested interface for the trace encoder. Additional fields may be necessary or more convenient for representing certain data in a way that is more convenient for a given implementation (e.g., indicating pc+2 or pc+4 directly).

```
1
2  package TraceEncoder;
3
4  typedef Bit#(ILEN) Instr;
5  typedef Bit#(XLEN) GPR;
6  typedef Bit#(FLEN) FPR;
7  typedef Bit#(XLEN) CSR;
8  typedef Bit#(5) RegAddr;
9  typedef Bit#(12) CsrAddr;
10 typedef Bit#(MLEN) MemAddr;
11 typedef Bit#(TMax#(XLEN,FLEN)) MemData;
12
13 typedef enum { ISIZE16BIT, ISIZE32BIT
14    } ISize deriving (Bits, Eq, FShow);
15
16 typedef enum { MEMSIZE8BIT, MEMSIZE16BIT, MEMSIZE32BIT,
17    MEMSIZE64BIT } MemSize deriving (Bits, Eq, FShow);
18
19 typedef struct {
20
21    Bit#(XLEN) pc;
22
23    // instruction
24    Maybe#(Tuple2#(ISize,Instr)) instr;
25
26    Maybe#(Tuple2#(RegAddr,GPR)) gpr;
27    Maybe#(Tuple2#(RegAddr,FPR)) fpr;
28
29    // CSRs, at most 4 necessary, for exceptions that update mtval
```

```
30     // (fetch, load, store, illegal instruction)
31     Vector#(4, Maybe#(Tuple2#(CsrAddr,CSR))) csr;
32
33     // unobserved and microarchitectural state
34     Maybe#(MemAddr) paddr;  // physical address
35     Maybe#(MemAddr) eaddr;  // effective address
36     Maybe#(MemSize,MemData) memdata;  // memory store data
37     Maybe#(CSR) fcsr;  // new fcsr
38
39     } TraceData deriving (Bits, Eq, FShow);
40
41
42 interface TraceEncoder_IFC;
43
44     interface Get#(Bit#(8)) out;
45
46     interface Put#(TraceData) in;
47
48     // superscalar processor could be supported as follows
49     //interface Put#(Vector#(N, Maybe#(TraceData))) in;
50
51     // unobserved state not associated with an instruction
52     interface Put#(Bit#(64)) mtime_in;
53     interface Put#(CSR) mcycle_in;
54     interface Put#(CSR) minstret_in;
55
56     // for RV32, include mcycleh and mcycle together, similar for minstret
57     //interface Put#(Tuple2#(CSR,CSR)) mcycle_in;
58     //interface Put#(Tuple2#(CSR,CSR)) minstret_in;
59
60 endinterface
61
62 endpackage
```

Appendix C

Examples

The following examples are based on a hart supporting rv64gc and Sv48. Thus, XLEN=64, FLEN=64, ILEN=32, and MLEN=64.

Note that these are simply examples, and in particular reflect a core that is operating properly.

Table C.1: `add gp, t0, t1`

Bytes	Meaning
01	begin group
03	increment pc
11	32-bit instruction
b3 81 62 00	instruction opcode
04	full register
03 10	gp (x3)
34 12 00 00 00 00 00 00	new gp data
02	end group

Table C.2: `jr 256(sp)`

Bytes	Meaning
01	begin group
07	additional state
10	pc
00 01 00 0c 00 00 00 00	new pc value
11	32-bit instruction
67 00 01 10	instruction opcode
02	end group

Table C.3: `fadd.s ft7, fa3, fa7`

Bytes	Meaning
01	begin group
03	increment pc
11	32-bit instruction
d3 f3 16 01	instruction opcode
04	full register
27 10	ft7 (f7)
f9 f4 fe 66 ff ff ff ff	new ft7 data (note: full FLEN register)
06	incremental register update (OR)
03 00	fcsr
01	fcsr bits to set (i.e., new floating point exceptions)
02	end group

Table C.4: `c.sub a0, a1`

Bytes	Meaning
01	begin group
03	increment pc
10	16-bit instruction
0d 8d	instruction opcode
04	full register
10 10	a0 (x10)
78 56 34 12 ff ff ff ff	new a0 data
02	end group

Table C.5: `lw tp, 8(t0)`

Bytes	Meaning
01	begin group
03	increment pc
10	32-bit instruction
03 a2 82 00	instruction opcode
04	full register
04 10	tp (x4)
aa 55 aa 55 aa 55 aa 55	new tp data
07	additional state
02	physical address identifier
08 00 00 01 00 00 00 00	physical address
02	end group

Table C.6: `csrrc s1, mie, a1`

Bytes	Meaning
01	begin group
03	increment pc
11	32-bit instruction
f3 b4 45 30	instruction opcode
04	full register
09 10	s1 (x9)
88 08 00 00 00 00 00 00	new s1 value
04	full register
04 03	mie
80 08 00 00 00 00 00 00	new mie value
02	end group

Table C.7: `00 00 00 00` (illegal instruction)

Bytes	Meaning
01	begin group
07	additional state
10	pc
00 00 01 00 00 00 00 00	new pc value
11	32-bit instruction
00 00 00 00	instruction opcode
04	full register
41 03	mepc
34 12 08 00 00 00 00 00	new mepc value
04	full register
42 03	mcause
02 00 00 00 00 00 00 00	new mcause value
04	full register
00 03	mstatus
b0 18 00 00 0a 00 00 00	new mstatus value
04	full register
43 03	mtval
00 00 00 00 00 00 00 00	new mtval value
07	additional state
01	privilege mode
03	machine mode
02	end group

Table C.8: interrupt

Bytes	Meaning
01	begin group
07	additional state
10	pc
00 00 01 00 00 00 00 00	new pc value
04	full register
41 03	mepc
56 12 08 00 00 00 00 00	new mepc value
04	full register
42 03	mcause
03 00 00 00 00 00 00 80	new mcause value
04	full register
00 03	mstatus
b0 18 00 00 0a 00 00 00	new mstatus value
04	full register
43 03	mtval
00 00 00 00 00 00 00 00	new mtval value
07	additional state
01	privilege mode
03	machine mode
02	end

Appendix D

Trace Data Size

The following table shows the trace data size for various instruction formats.

The current tandem verification information takes 38 bytes for each instruction. Currently, floating point support is broken. With floating point support fixed, each instruction will take 46 bytes.

The proposed trace data format takes fewer bytes for each instructions than the existing tandem verification format in the majority of cases. Additionally, for the cases that the proposed trace format takes more bytes, this is primarily because the proposed trace provides additional information (all registers on exceptions, physical and effective addresses).

Instruction Category	Fields	Size (bytes)
integer computation	pc, instr, gpr	23
nop	pc, instr	12
branch, jump	pc, instr	27
branch and link, jump and link	pc, instr, gpr	32
load	pc, instr, gpr, paddr	33
load (virtual address)	pc, instr, gpr, paddr, eaddr	43
store	pc, instr, paddr, memdata	32
store (virtual address)	pc, instr, paddr, eaddr, memdata	42
sc, amo	pc, instr, gpr, paddr, eaddr, memdata	53
csr read-write	pc, instr, csr, gpr	34
csr read only	pc, instr, gpr	23
csr write only	pc, instr, csr	23
floating point computation	pc, instr, fpr, fcsr	27
floating point move	pc, instr, gpr	23
floating point conversion	pc, instr, gpr, fcsr	27
floating point load	pc, instr, fpr, paddr, eaddr	43
floating point store	pc, instr, paddr, eaddr, memdata	42
interrupt	pc, xepc, xcause, xstatus, xtval, priv	59
exception	pc, instr, xepc, xcause, xstatus, xtval, priv	68

Appendix E

System Diagram

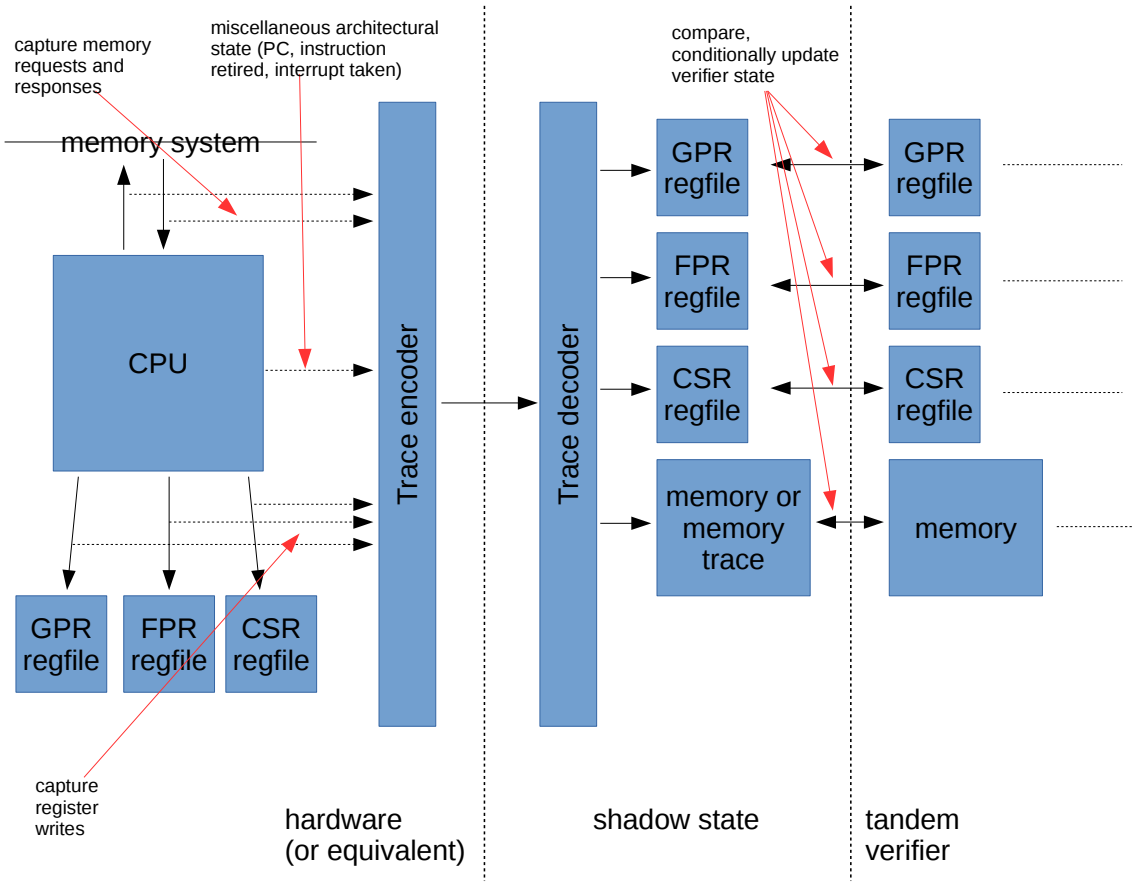

Figure E.1: Trace and Tandem Verification System Diagram

Appendix F

Change Log

Revision	Date	Author(s)	Description
2e4f73b	2019-03-07	rsnikhil	Updated Show_Trace_Data_Haskell/src/Parse_Trace_Data.hs to be a truly lazy list
e00978d	2019-03-04	rsnikhil	Updated Show_Trace_Data (C version) to Dec 20 2019 spec. Created Show_Trace_Data_Haskell (Haskell version).
4f95627	2018-12-30	rsnikhil	Fixed Show_Trace_Data/show_trace_data.c to forgive an absent 'mem_rsp' after a 'mem_req' in STATE_INITs.
b1d1aa9	2018-11-20	Darius Rad	Bump the version.
874697d	2018-11-20	Darius Rad	Clarified packing of fields smaller than one byte.
cee550d	2018-11-06	rsnikhil	Fixed an elf_to_trace.c bug for MLEN=34
707b2ad	2018-11-04	rsnikhil	In show_trace_data.c, added printouts for orphan end_groups and nested begin_groups
9760f4f	2018-10-15	rsnikhil	Completed show_trace_data.c; added elf_to_trace.c to convert and ELF file into a memory-loading trace file
58c456b	2018-10-15	Darius Rad	Clarify that memory responses are not required.
8acd60d	2018-09-20	Darius Rad	Rename microarchitectural state to additional state. Add additional state identifier for pc.
a7bc25f	2018-10-13	rsnikhil	Now uses RV32/RV64 compiler macros to fix XLEN. Makefile has RV32 and Rv64 targets.
61c0bc5	2018-10-09	rsnikhil	Added Show_Trace_Data with program to display a trace data file in human-readable form
fd22d50	2018-09-18	Darius Rad	Traps change privilege mode (in general).
91d4bc6	2018-09-18	Darius Rad	xTVAL is updated on all traps.
418a1c0	2018-09-18	Darius Rad	Use the term 'effective' instead of 'virtual' address for supervisor and user mode addresses.
ec391f0	2018-09-13	Darius Rad	Fix length of MLEN fields and clarify value of MLEN.
76c9346	2018-09-12	Darius Rad	Initial import to TeX of trace protocol specification.