The 4th lowRISC Release: Tagged Memory and Minion Cores

Wei Song, Jonathan Kimmitt, Alex Bradbury, and Robert Mullins
University of Cambridge / lowRISC

10 May 2017

lowRISC

- lowRISC: A not-for-profit organisation based in Cambridge, UK.

- We produce open-source and free Linux capable SoC platforms
  - Open source from the core to the on-chip interconnects (and any IPs if available)
  - Free for both academic and commercial uses
  - 64-bit multicore (Rocket + PULP)
  - SystemVerilog top level
  - tagged memory and minion cores

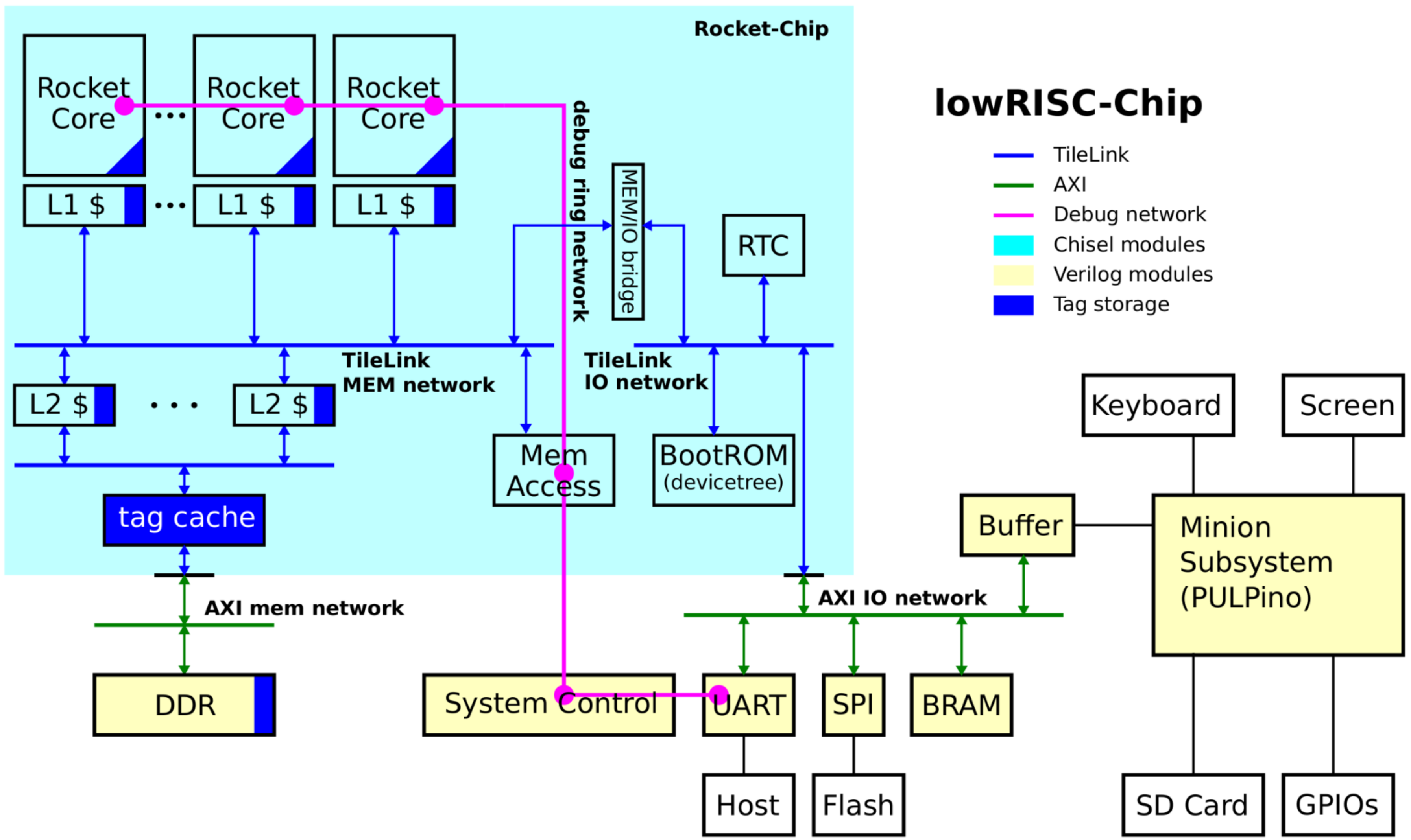
- Open development
  - Share as much as possible
  - encouraging community effort
  - Regular tape-out with community contribution

- Aim to be the “Linux of the hardware world”

Our Releases

- lowRISC with tagged memory, April 2015
  - Initial support for read/write tags.
- Untethered lowRISC, December 2015
  - A standalone SoC without the companion ARM core.
- lowRISC with a trace debugger, July 2016
  - First implementation of a debug infrastructure.
  - A trace debugger to collect instruction and software defined traces.
- **lowRISC with tagged memory and minion core, May 2017**
  - Bring back tagged memory with built-in tag manipulation and check in the core pipeline with an optimised tag cache.
  - A full SD interface using a reduce PULPino as a minion core.
- Improved tagged memory and minion cores
  - Improve the support for both tagged memory and minion cores.
  - Merge update from upstream (interrupt controller, run-control debugger and TileLink2).
  - Adopt a regular release cycle.

Overall Structure



General-Purpose Tagged Memory

- Implementation
  - Associate tags (metadata) with each physical memory location.
  - Tags are stored in on-chip caches and a tag cache is inserted between the LLC and main memory.
  - Built-in support of tag manipulation and check in the core pipeline.

- Potential use cases
  - Protection of code pointers
  - Hardware-assisted control-flow integrity
  - Infinite hardware memory watch-points
    - E.g. canaries on stack
  - Poisoning (simple IFT)
    - Ensure sensitive data does not leak

Problems of the Previous Tag Cache

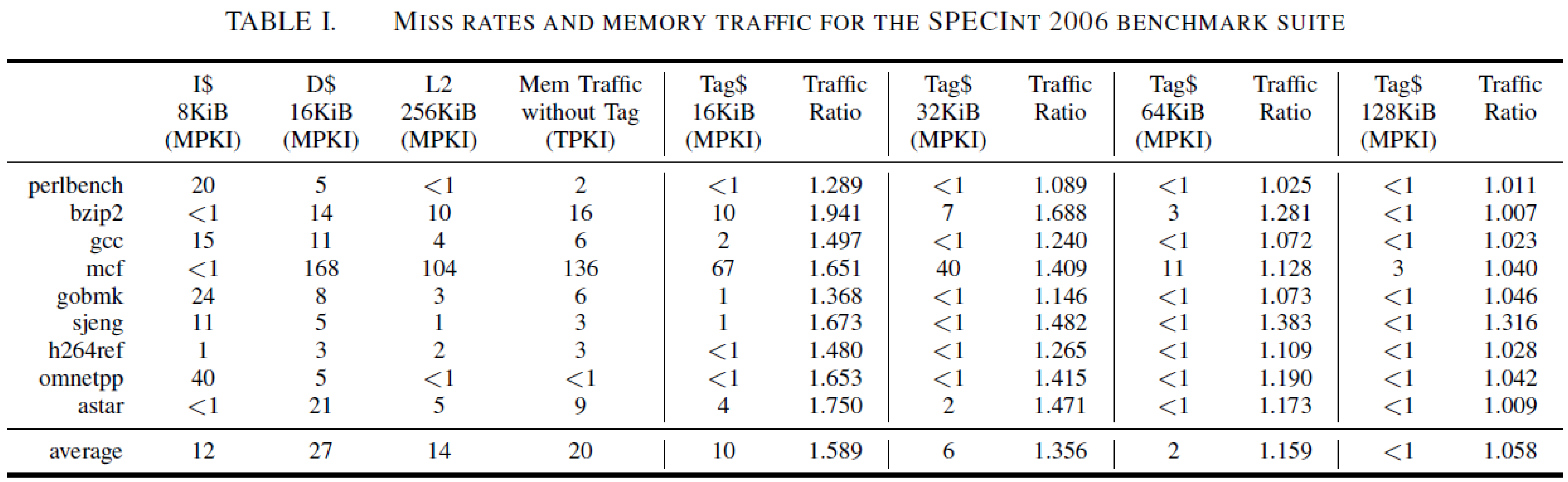


TABLE I. MISS RATES AND MEMORY TRAFFIC FOR THE SPECINT 2006 BENCHMARK SUITE

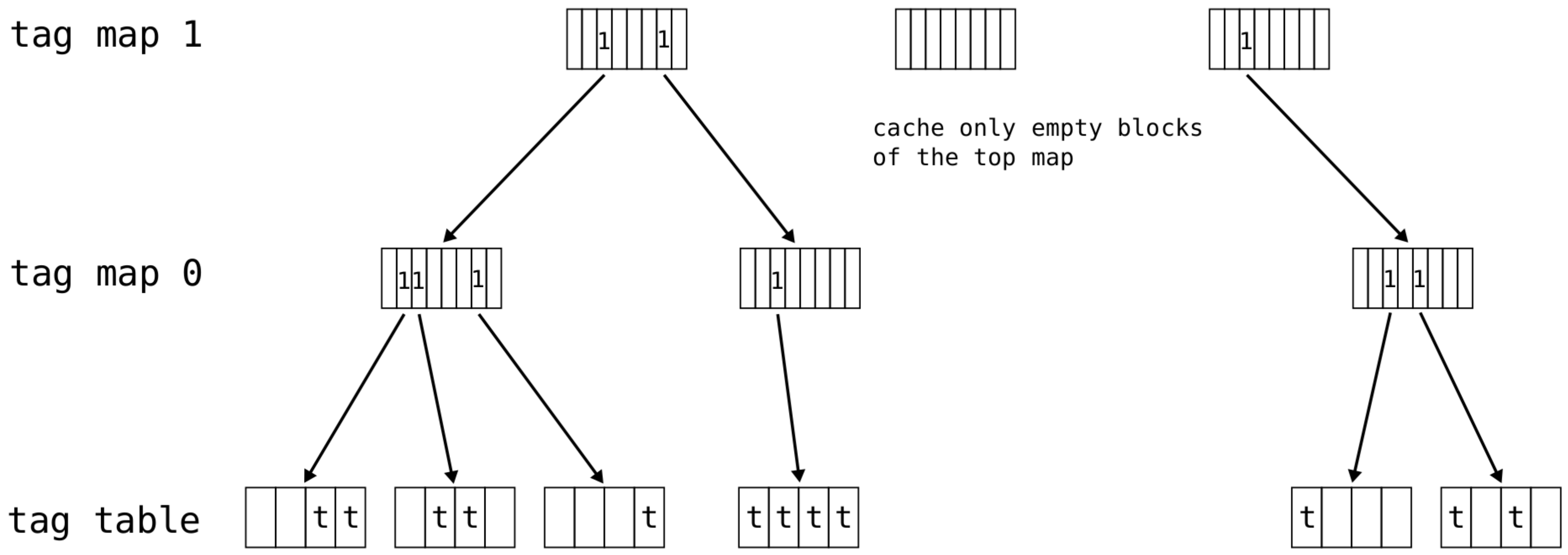
	I$ 8KiB (MPKI)	D$ 16KiB (MPKI)	L2 256KiB (MPKI)	Mem Traffic without Tag (TPKI)	Tag$ 16KiB (MPKI)	Traffic Ratio	Tag$ 32KiB (MPKI)	Traffic Ratio	Tag$ 64KiB (MPKI)	Traffic Ratio	Tag$ 128KiB (MPKI)	Traffic Ratio
perlbench	20	5	<1	2	<1	1.289	<1	1.089	<1	1.025	<1	1.011
bzip2	<1	14	10	16	10	1.941	7	1.688	3	1.281	<1	1.007
gcc	15	11	4	6	2	1.497	<1	1.240	<1	1.072	<1	1.023
mcf	<1	168	104	136	67	1.651	40	1.409	11	1.128	3	1.040
gobmk	24	8	3	6	1	1.368	<1	1.146	<1	1.073	<1	1.046
sjeng	11	5	1	3	1	1.673	<1	1.482	<1	1.383	<1	1.316
h264ref	1	3	2	3	<1	1.480	<1	1.265	<1	1.109	<1	1.028
omnetpp	40	5	<1	<1	<1	1.653	<1	1.415	<1	1.190	<1	1.042
astar	<1	21	5	9	4	1.750	2	1.471	<1	1.173	<1	1.009
average	12	27	14	20	10	1.589	6	1.356	2	1.159	<1	1.058

- Motivation
  - A simple set-associative cache is inefficient as most cached tags are unset.
  - Most data are usually untagged (with unset tag).
  - Applications which do not use tags should not suffer.

Optimised Tag Cache

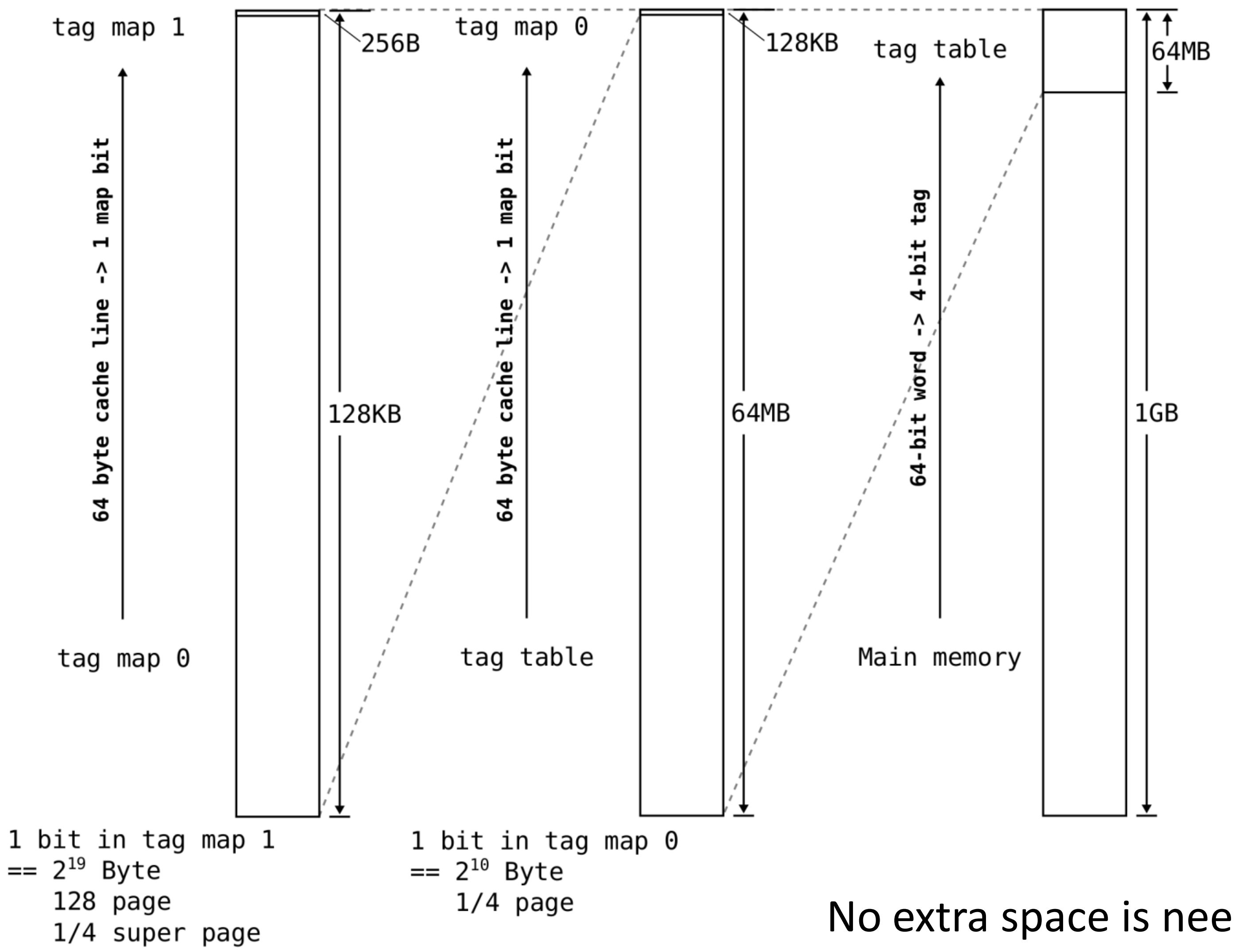
- Solution
  - Compress the cached tags using multiple levels of bit maps
    - A cache line size of unset tag is stored as a 1-bit flag.
    - Avoid cache lines of unset tags in the tag cache.
  - Avoid fetching or writing back empty tag cache lines.
  - zero overhead for applications that do not use tags.
  - Improve cache efficiency.
    - A small tag cache is enable to cover a large memory space.

Logical View of the Tag Partition



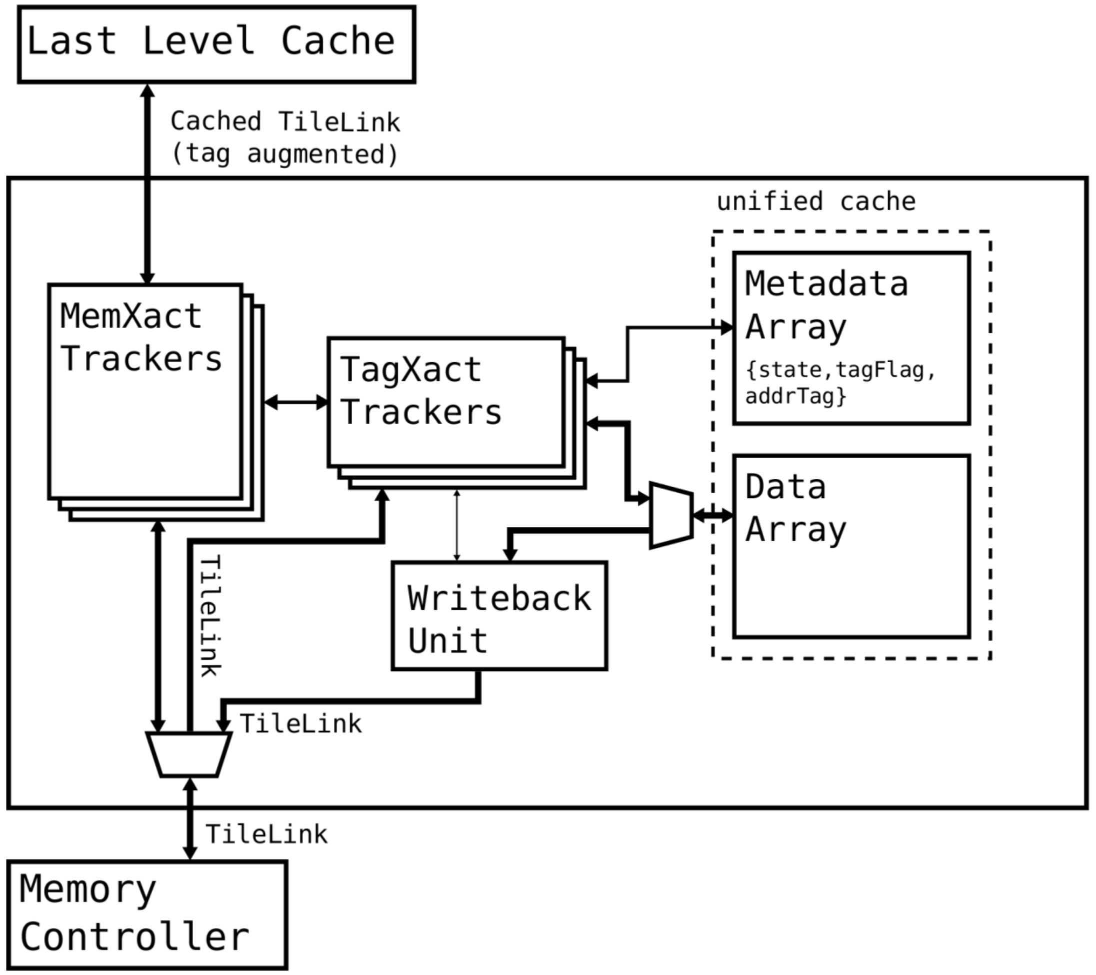
Node	A cache line size of data in the tag partition.
Tag table	Nodes of actual tags.
Tag map 0	Nodes of bit maps that map a tag table node into a 1-bit flag.
Tag map 1	Nodes of bit maps that map a tag map 0 node into a 1-bit flag.

Physical View of the Tag Partition



No extra space is needed for tag maps.

Structure of the Tag Cache



Metadata and Data array
Unified tag cache for all levels of tag cache lines (map and table).

MemXact Tracker
Parallel tracker to handle multiple simultaneous memory accesses from the last-level cache.

TagXact Trackers
Parallel trackers to handle an access to the unified cache array.

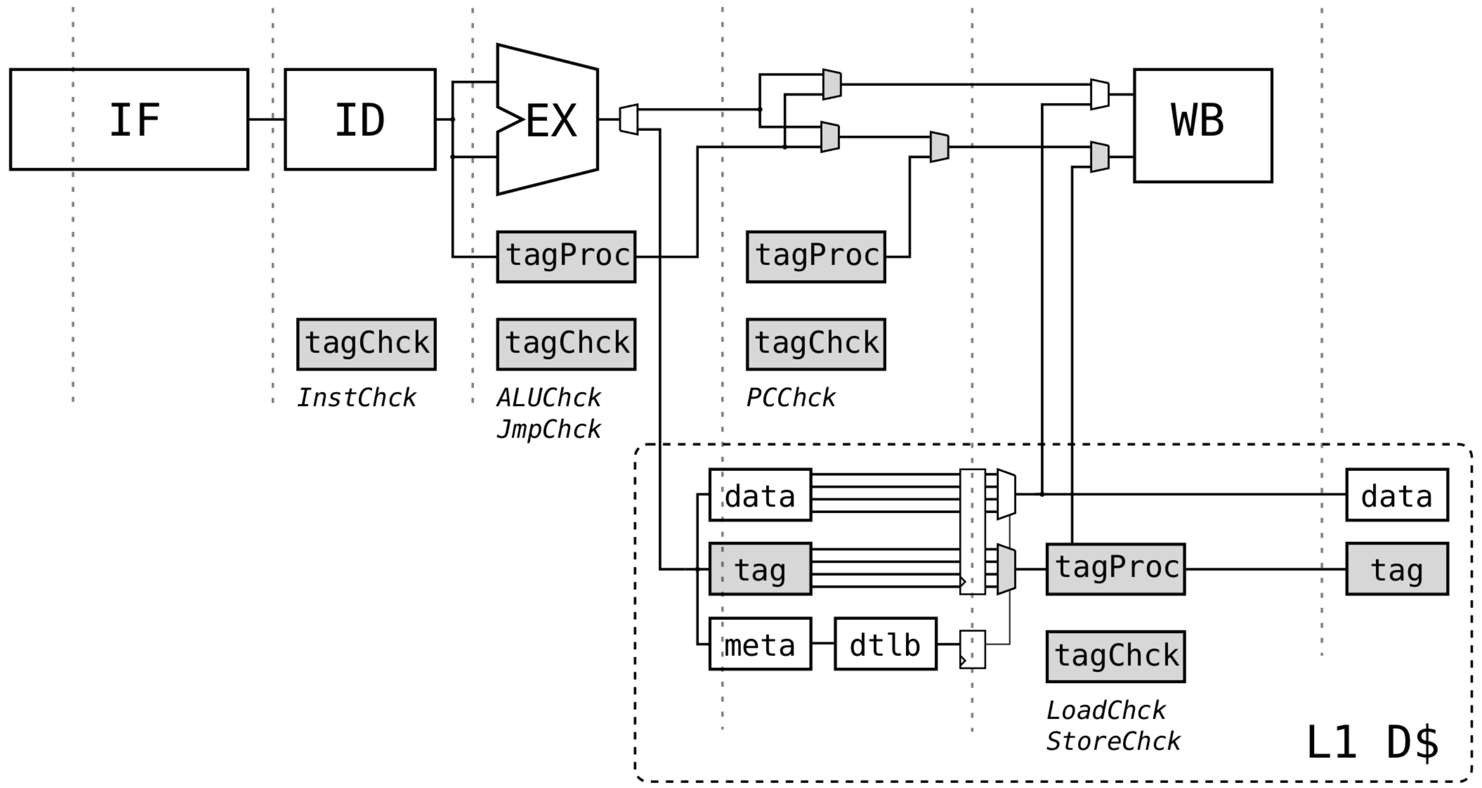
Writeback Unit
A shared writeback unit for evicting dirty and nonempty tag cache lines.

Concurrent Transaction Control

- Maintain consistency between map and table nodes
  - A memory transaction may temporarily break the consistency (non-atomic updating of map bit).
  - An access to the unified tag cache array must be atomic.
  - Block a memory transaction until related map bits return a consistent state.

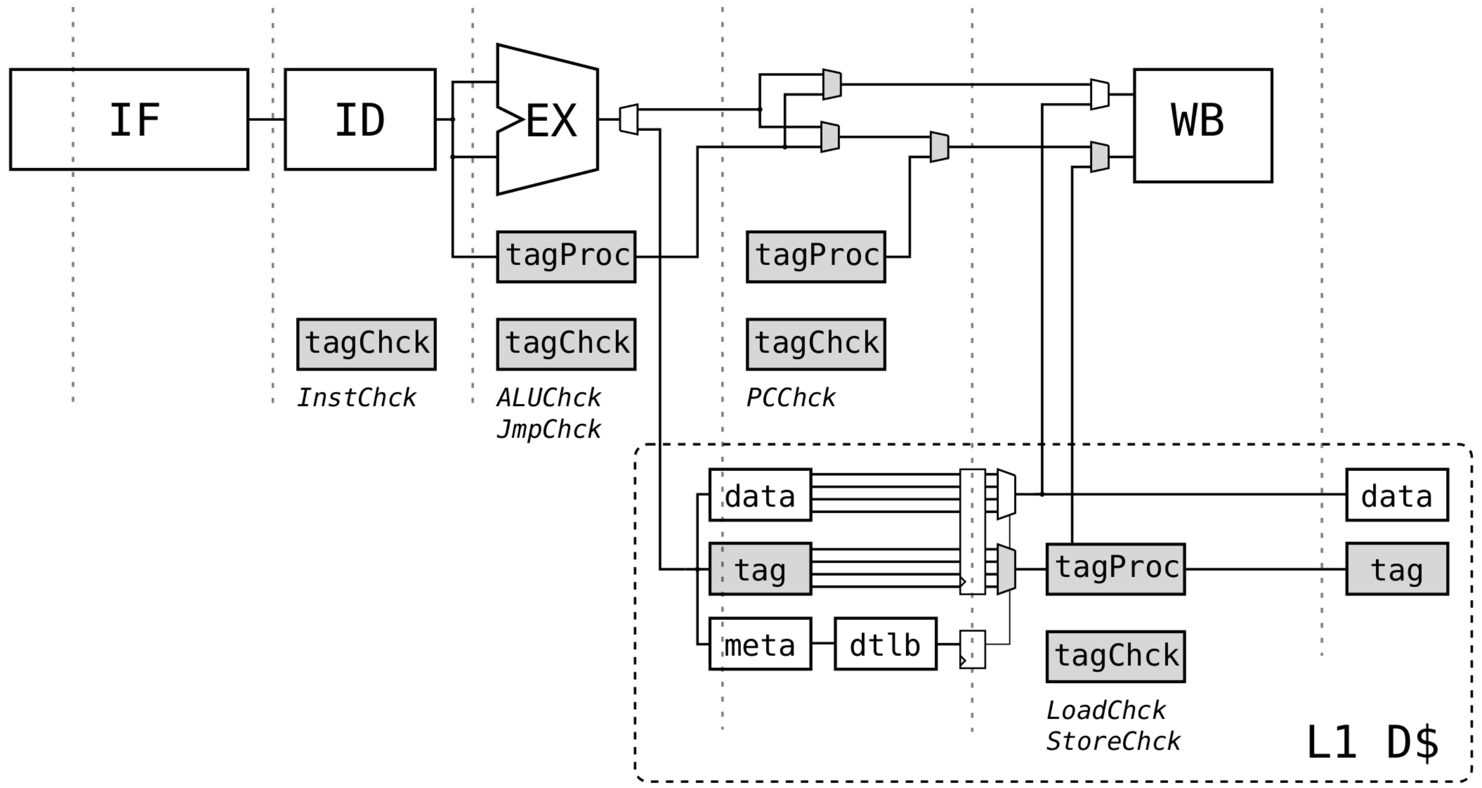
- Other optimisation
  - Bottom-up search order: always search table nodes first.
  - Create instead of fetch for empty lines.
  - Avoid writing back empty lines unless it is top map node.

Tag Support in Core Pipeline



ID	Check for instruction tags.
EX	Tag manipulation along with ALU operations.
	Check the tags of source registers for ALU and jump instructions.
MEM	Propagate tags to link registers.
	Check the instruction tags of jump targets.

Tag Support in Core Pipeline (cont.)



D$ Store tags along side with data
Propagate tags from memory to register file and vice versa.
Check the memory tags for load or store operations.

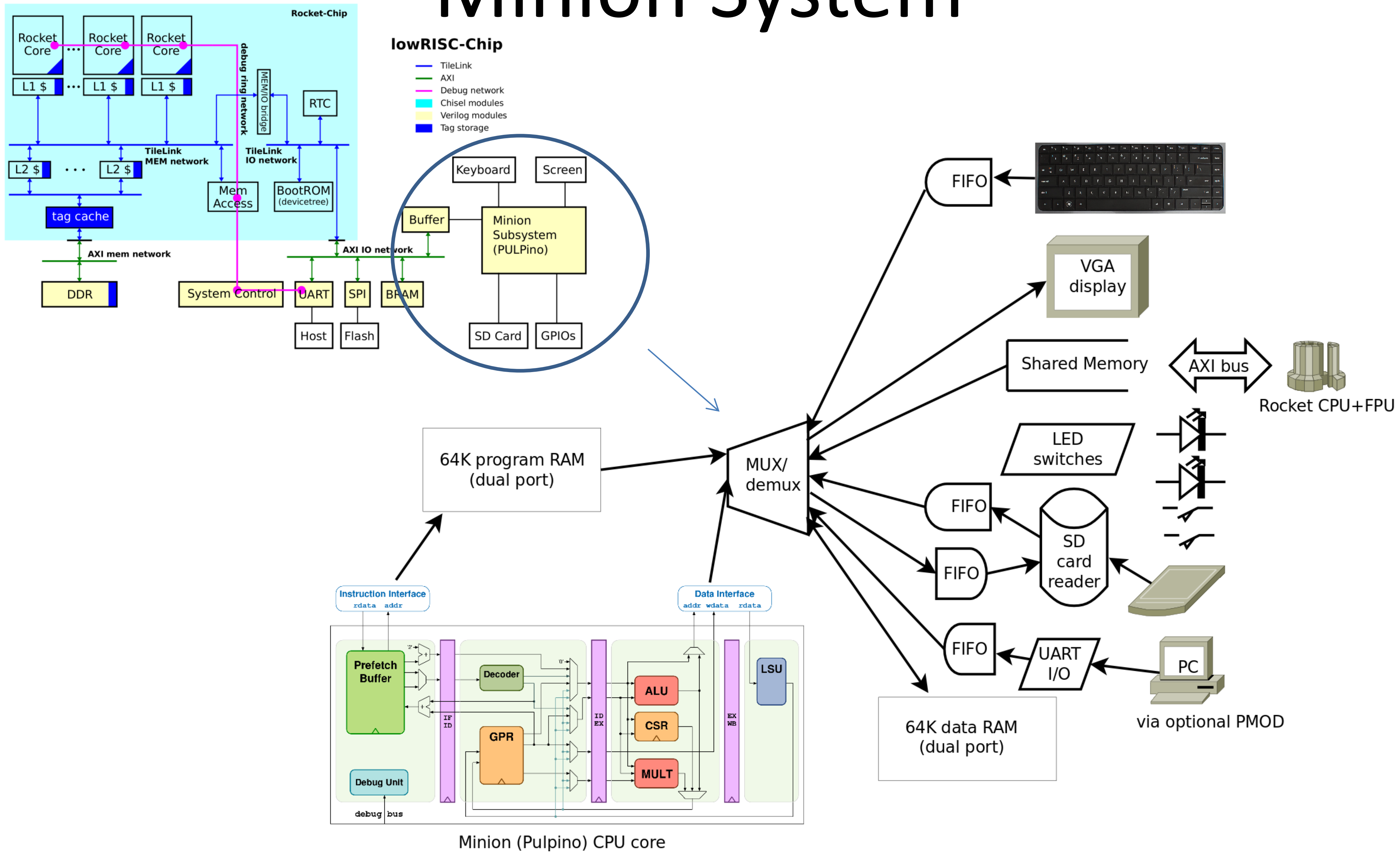
Special Instructions and CSRs

- Tag read and write
  - **TAGR** rd, rs1
    (rd_t, rd) <= (0, rs1_t)
  - **TAGW** rd, rs1, imm
    (rd_t, rd) <= (rs1+imm, rd)
- Tag control CSR
  - **mtagctrl** (tagctrl)
    A set of masks for each tag function
  - **stagctrl**, **mstagctrlen**
    tagctrl <= (stagctrl & mstagctrl) | (tagctrl & ~mstagctrl)
  - **utagctrl**, **mutagctrlen**
    tagctrl <= (utagctrl & mutagctrl) | (tagctrl & ~mutagctrl)
- Tag extension in CSRs
  - mepc, sepc, mscratch, sscratch, mtvec, stvec

An Example Use-Case

- Code pointer protection
  - Mark valid code pointers .
  - Prevent them being overwritten with arbitrary data.
    - 1-bit CPTR tag
    - Allow load CPTR
    - Disallow store CPTR (prohibit overwrite code pointer)
    - Check jump target with CPTR (only jump to valid code pointer)
    - Check indirect jump's rs1 with CPTR (only allow valid indirectional jump)

Minion System



Minion Driven Full SDHC

- Communication between minion and Rocket
  - A shared 64KB dual port memory
  - No coherence control yet.
- SDHC interface
  - Support automatic speed detection (5MHz by default).
  - Support mounting SD inside Linux
  - Support read and write operations.
  - Run the SD driver on Rocket (will move it to minion)

Flexible Boot Procedure

- Standard-alone
  - FPGA starts from Flash
  - Initial bootloader reads BBL+Linux from SD through the minion core.
- Program through Vivado
  - Vivado configs FPGA
  - Initial bootloader reads BBL+Linux from SD throgh the minion core.
- Boot from trace debugger
  - FPGA is configured by Flash or Vivado
  - Load BBL+Linux to DDR using trace debugger (bypass the minion core)
  - Jump to DDR memory

Other Development in lowRISC

- QEMU minion emulation
  - Emulate a Rocket core using QEMU while connecting to a real minion core on FPGA.
  - Jonathan Kimmitt is leading this effort.

- RISC-V LLVM Compiler
  - We plan to use it for a number of tagged memory use-cases.
  - We are producing a well documented “reference” backend for RISC-V
  - Alex Bradbury is leading this effort
    - >90% of GCC torture test suite is passing (RV32I). Basic support by end Q2
    - Full support, incl. ISA variants, est. Q4’17.

- Google Summer of Code 2017

Access to the New Release

- Source code
  https://github.com/lowrisc/lowrisc-chip/tree/minion-v0.4
- Tutorial (come out soon)
  http://www.lowrisc.org/docs/minion-v0.4/
- lowRISC
  http://www.lowrisc.org
- Mail list
  lowrisc-dev@lists.lowrisc.org

Thank You!