UNIVERSITY OF CAMBRIDGE

lowRISC
OPEN TO THE CORE

lowRISC: An Opensourced SoC Provider Based on RISC-V Rocket Cores

Wei Song

University of Cambridge / lowRISC

Content

- Introduction to RISC-V
- Introduction to lowRISC
- lowRISC SoC and the internals of the Rocket core
- Tagged memory
- Minion core
- Trace debugger

What is RISC-V

- 5th generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
- Experiencing rapid uptake in both industry and academia
- Standard maintained by non-profit RISC-V Foundation
- Both proprietary and open-source core implementations
- Supported by growing shared software ecosystem
- Appropriate for all levels of computing system, from microcontrollers to supercomputers

Krste, Mon1045-Why-RISC-V-ISAs-Want-to-be-Free

RISC-V ISA standard extensions

- Four base integer ISAs
  - RV32E, RV32I, RV64I, RV128I
  - RV32E is 16-register subset of RV32I
  - Only <50 hardware instructions needed for base
- Standard extensions
  - M: Integer multiply/divide
  - A: Atomic memory operations (AMOs + LR/SC)
  - F: Single-precision floating-point
  - D: Double-precision floating-point
  - G = IMAFD, “General-purpose” ISA
  - C: Compressed 16-bit instruction
- All the above are a fairly standard RISC encoding in a fixed 32-bit instruction format
- Above user-level ISA components frozen in 2014
  - Supported forever after

Krste, Mon1045-Why-RISC-V-ISAs-Want-to-be-Free

RISC-V foundation

- Mission statement

  "*to standardize, protect, and promote the free and open RISC-V instruction set architecture and its hardware and software ecosystem for use in all computing devices*."

- Established as a 501(c)(6) non-profit corporation on August 3, 2015
- Rick O'Connor recruited as Executive Director
- First year, 41+ "founding" members. Now ~60 members.

Google, IBM, Oracle, Qualcomm, Sumsung, nVidia, Microsoft, AMD, HP, NXP, Micron, Microsemi

Huawei, Andes, ICT, VeriSilicon, MediaTek, C-Sky

Available RISC-V Implementations

Rocket	UC Berkeley	ASIC	RV32/64GC	BSD	Chisel	A7/A53
BOOM	UC Berkeley	ASIC	RV64G	BSD	Chisel	A15/A57
Shakti	IIT-Madres	ASIC	RV32/64	BSD	Bluespec	All
RIVER	GNSS	ASIC/FPGA	RV64IMA	BSD	VHDL	A7
PULPino	ETH Zurich	ASIC	RV32I	Solderpad	SV	MCU
Orca	VectorBlox	FPGA	RV32IM	BSD	VHDL	MCU
SCR1	Syntacore	ASIC	RV32I	Solderpad	SV	MCU

Now let's talk about lowRISC…

Co-funders of lowRISC

- Robert Mullins
  - Asynchronous circuit: pausible clock demystify
  - NoC: Single-cycle VC router
  - Co-funder of Raspberry Pi
  - lowRISC and Loki
- Alex Bradbury
  - Contributor for Raspberry Pi
  - LLVM compiler support for Loki
  - Maintainer of LLVM weekly news
  - RISC-V LLVM port
- Gavin Ferris
  - Dreamworks, Radioscape (co-founder)
  - Aspect Capital (former CIO)
  - Angel donor

What is lowRISC

- lowRISC is a not for profit organization from the University of Cambridge.
- Objectives
  - Provide opensourced and free system-on-chip (SoC) platforms
  - Linux capable
  - Highly customizable
  - Multiple application cores (Rocket/BOOM)
  - Multi-level coherence cache hierarchy
  - Minion cores as IO processors and hardware accelerators
  - General purposed tagged memory (security, debug and performance improvement)

Why do we want to do it?

- Most commercial ISAs and microarchitectures are not free
  - Extra cost for small companies doing chips
  - Barrier for academic research on microarchitectures
- Nearly impossible to customize an ISA or microarchitecture
  - Overly expensive architectural license (ARM)
  - Complicated customizing procedure with opaque evaluation process (Tensilica/ARC)
  - Per configuration license/royal fee.

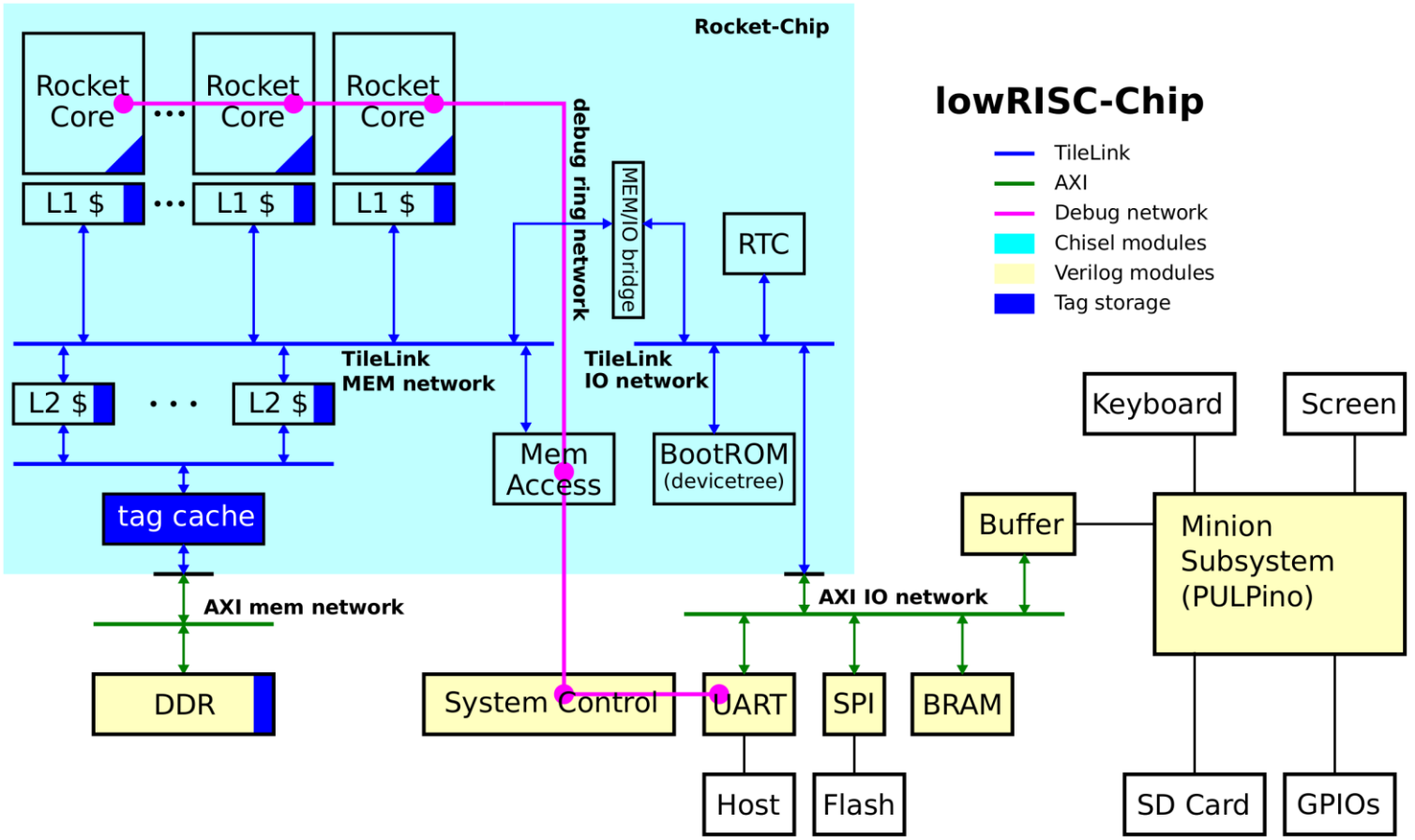
What is our approach?

- RISC-V, Rocket and PULPino
  - Open and free from ISA to implementation
  - Both Rocket and PULPino are tape-out verified
  - Strong community and industry support
- lowRISC SoC Platform
  - SystemVerilog top level and interconnects
  - Encapsulated Chisel islands of Rocket and caches.
- Community involvement
  - Fully open development environment
  - Encourage community involvement
  - Hope to have regular tape-outs with community contributed blocks

Release history of lowRISC

- 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 reduced 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.

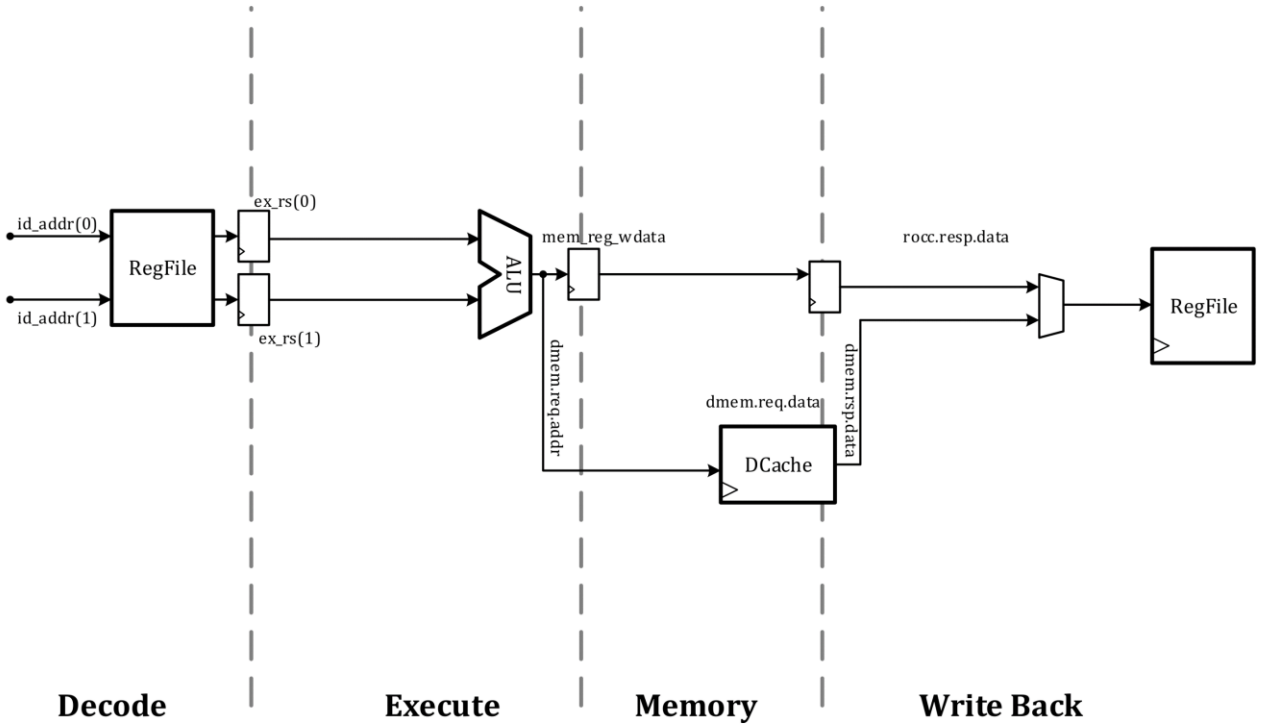
Current SoC structure



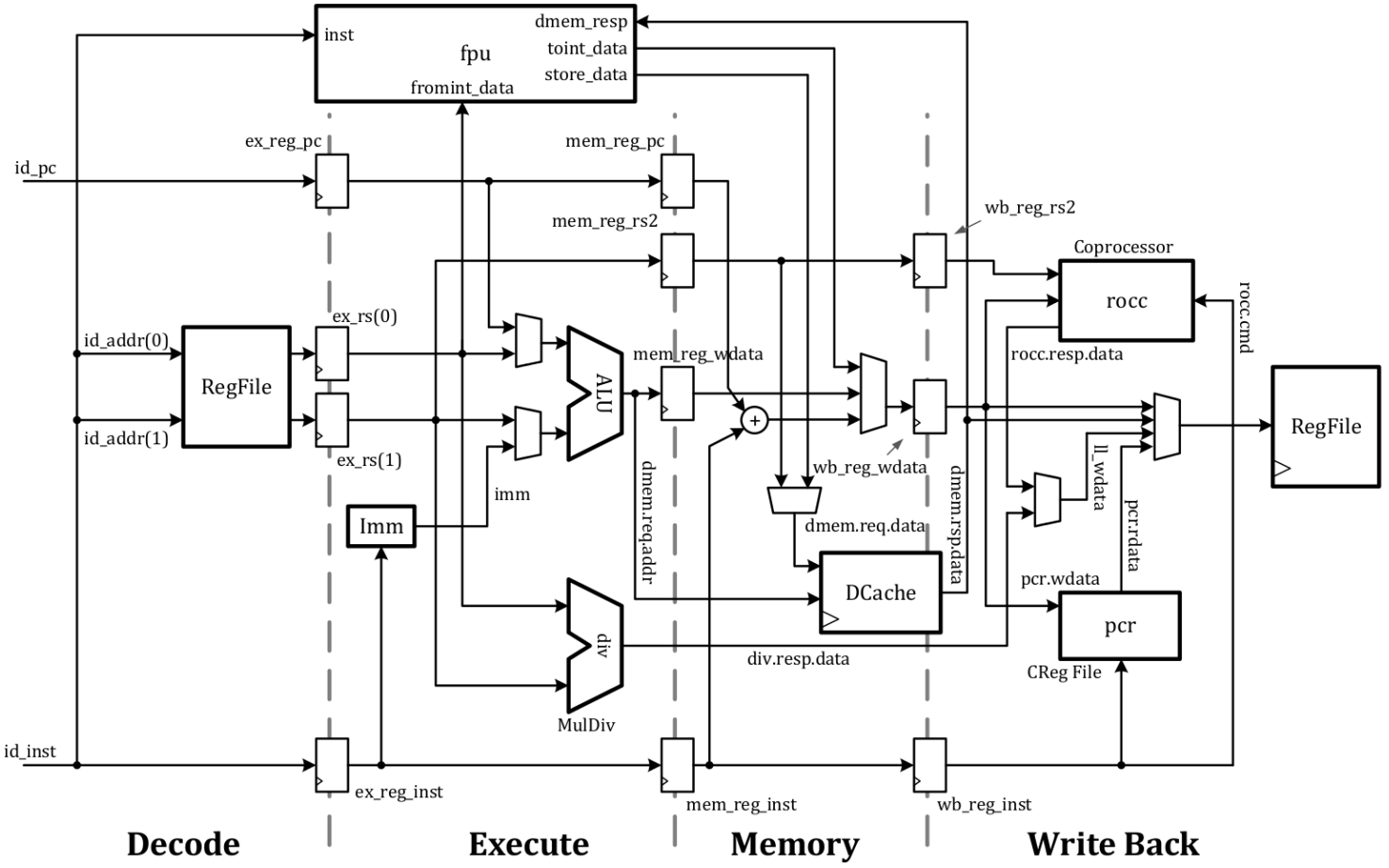
Summary of current SoC

- Rocket chip
  - Chisel code (Summer 2016)
  - Tilelink with L2
  - Privileged spec ~ 1.9.1
  - GCC and Linux Kernel (End of 2016)
- lowRISC extra
  - SystemVerilog top level, AXI 4 interconnects
  - Trace debugger
  - Builtin tagged memory support
  - Full SD interface using a PULPino core
  - Makefile scripts and FPGA demos (Nexys4-DDR)

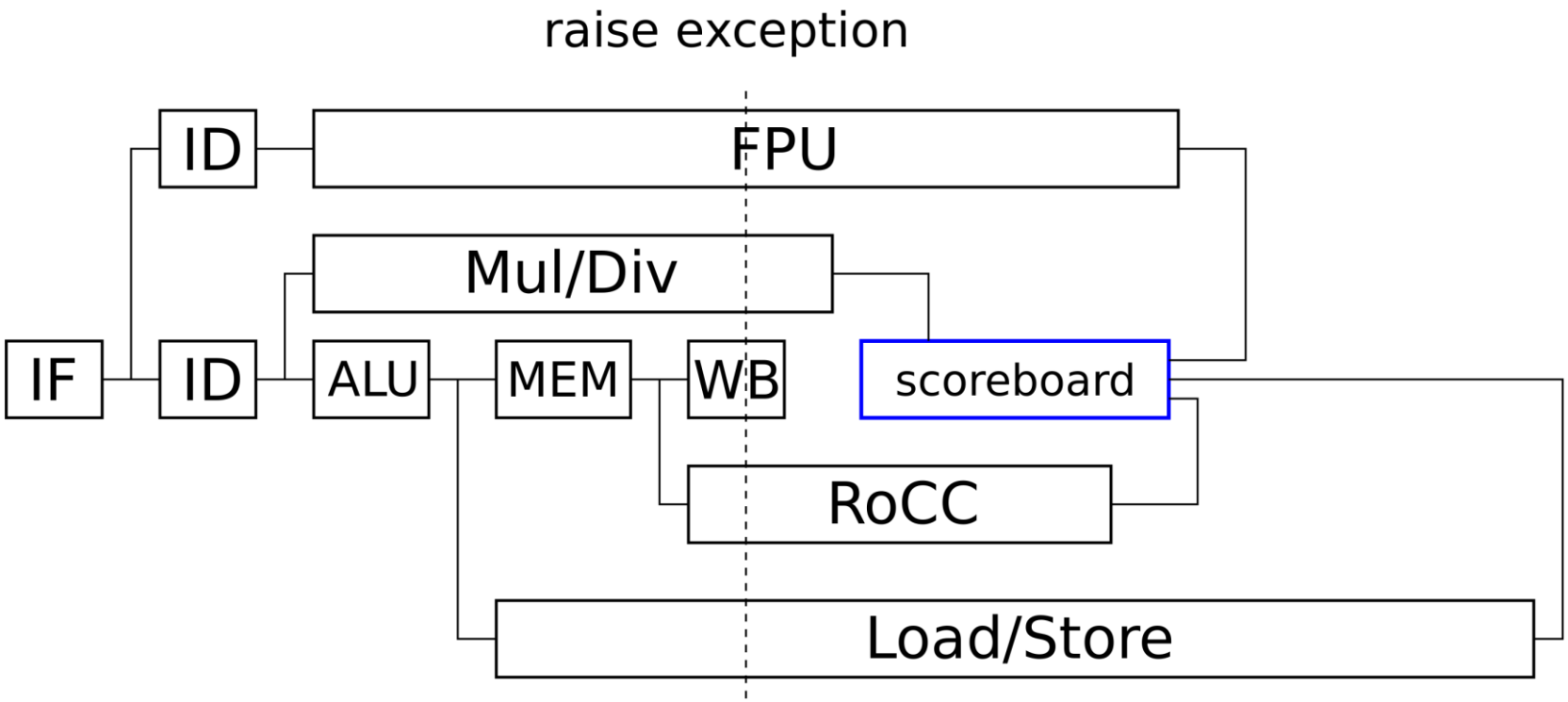
Rocket core pipeline



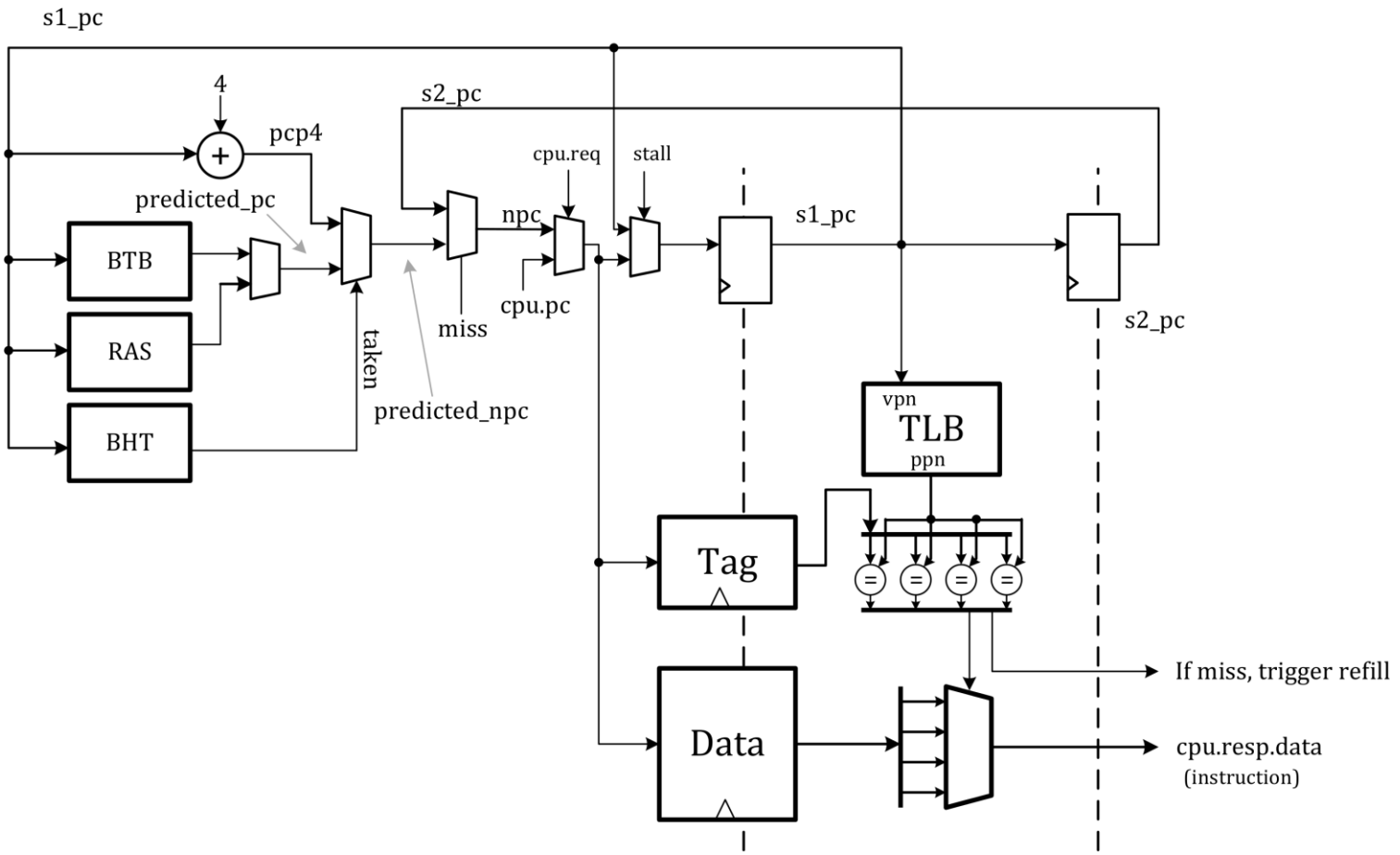
Rocket core pipeline



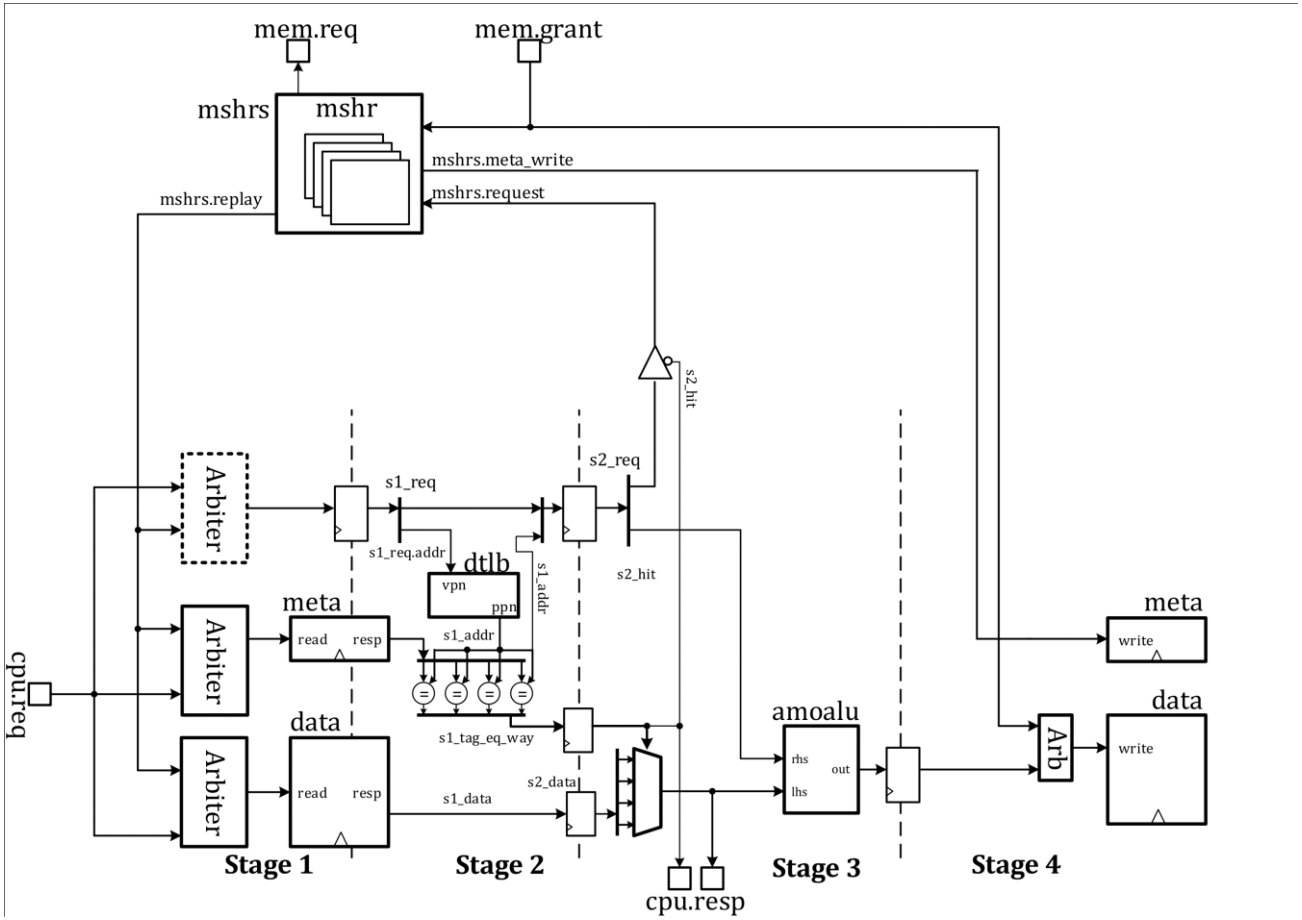
In-order commit but out-of-order RF update



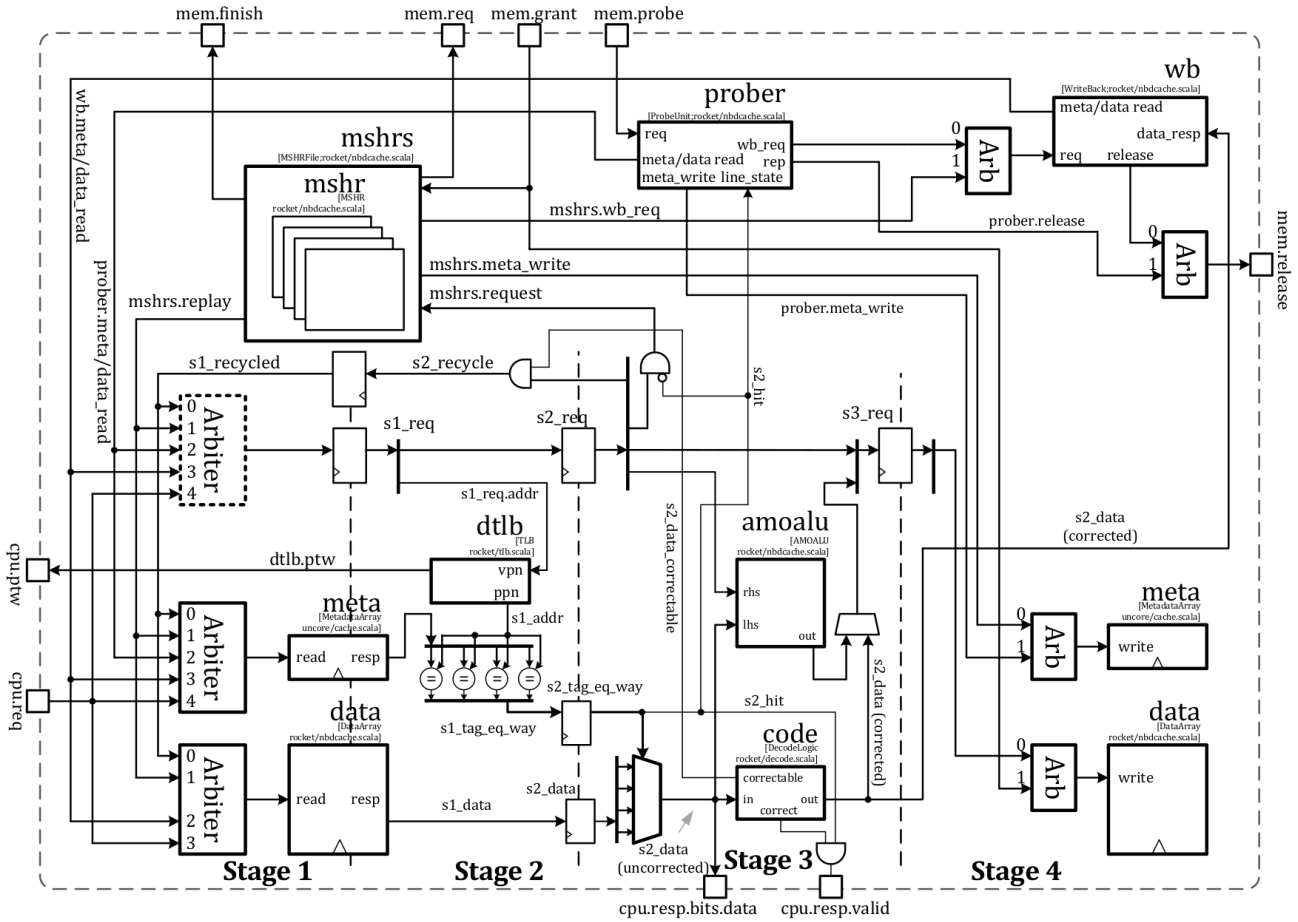
Instruction fetch and cache



Nonblocking data cache



Nonblocking data cache



Boot procedure

- First stage boot ROM
  - 0x00000000
  - Automatically generated by Chisel compiler
  - Device map (DTS), jump to boot RAM
- Second stage boot RAM
  - 0x40000000
  - Compiled by GCC, load by bitstream/debugger
  - Copy BBL/Kernel to DDR (drive SD), jump to BBL
- Berkeley bootloader (BBL)
  - 0x80000000
  - Compiler by GCC, load by second stage bootloader/debugger
  - Initialise I/O peripherals, virtual memory, load kernel to virtual space
  - Jump to Kernel and switch to S mode
- Linux kernel
  - Run in S mode, virtual memory space
  - Access I/O peripherals through SBI (also drivers if added)

Our special features

- Tagged memory
  - General purposed tagged memory support
  - Hardware-assisted secure system
- Minion cores
  - IO processor
  - Performance monitor
  - Accelerators
- Trace debugger
  - Non-intrusive trace collection
  - Debugging real-time multicore systems

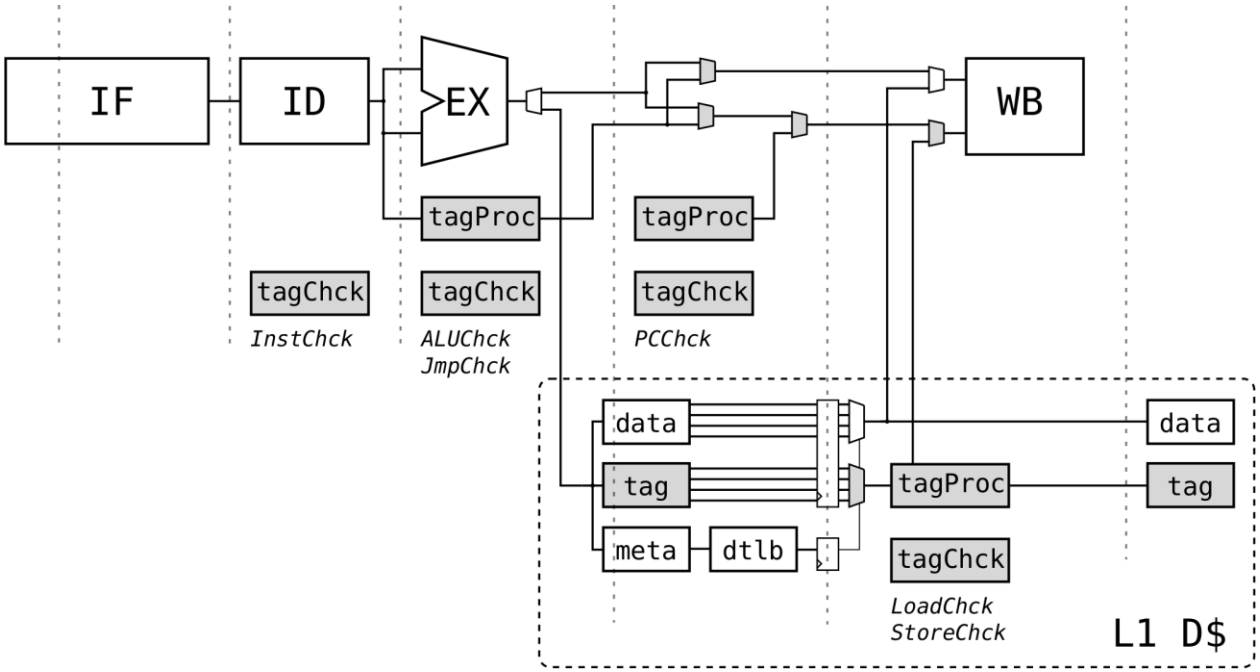
Tagged memory

- Tagged memory is not new
  - Lisp machine (60s) uses tags to type data structures
- Why bring it back? For security.
  - Fine granularity at word level
    - Page-table and trust-zone
  - Security is a critical issue nowadays
    - ROP attack
    - Cloud computing
    - IoT
  - Cost of extra transistors / memory is tolerable.

Tagged memory

- What is a tag?
  - A metadata (>= 4-bit) attached to every addressable double-word (64-bit)
- What is tagged?
  - 0x40000038 ADD r4, r5, r6
    - Instruction, rs1, rs2, rd, pc
  - 0x4000003c LD r4, 8(r4)
    - Memory @ r4+8
  - 0x40000040 MCALL
    - mepc, mtvec, mscratch (also for S mode)
- General description
  - (rd, mem, xcpt) <= func(instr, pc, rs1, rs2, mem)
  - (rd_t, mem_t, xcpt) <= funct($instr_t$, pc_t, $rs1_t$, $rs2_t$, mem_t)

Builtin tag support in Rocket core



Use tags for debug breakpoint

		tag	
	0x4000622c	4'b0000	LD t5, 4(sp)
Breakpoint 1	0x40006230	4'b0101	LD t6, 8(sp)
Breakpoint 2	0x40006234	4'b0101	ADD t5, t5, t6
	0x40006238	4'b0000	ADDI t5, t5, -16
	0x4000623c	4'b0000	SD t5, 4(sp)
	0x40006240	4'b0100	SD t0, 8(sp)
Breakpoint 3	0x40006244	4'b0100	SRET

Support infinite breakpoints.

Use tags for data watchers

	tag		
	4’b0011	0xffffffff_ffe00478	buffer
	4’b0011	0xffffffff_ffe00470	buffer
local stack	4’b0011	0xffffffff_ffe00468	buffer
	4’b0011	0xffffffff_ffe00460	local variable
	4’b0011	0xffffffff_ffe00458	local variable
canary	4’b0000	0xffffffff_ffe00450	canary
protect return addr.	4’b0000	0xffffffff_ffe00448	return address
const. argument	4’b0001	0xffffffff_ffe00440	argument

stack

4’bxxx1: readable 4’bxx1x writable
Protect from stack overflow attack.

Use tags for control flow integrity

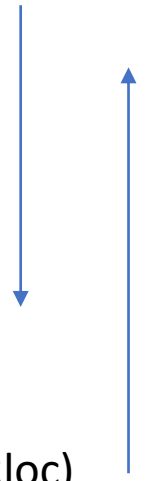
- Check the instruction tag of the branch target is legal.
  - Forward CFI: only jump to legal function subsets
  - Backward CFI: only return to legal locations (ROP)

```
            LD    t5, 48(sp)      ; propagate tag 4'b0001 from @(sp+48)
            JALR ra, t5, 8        ; set pc tag to 2'b01, set tag of ra to 4'b0001
Rloc:       ADD t2, t1, t5

…….

Func:       PUSH ra               ;  check inst tag == pc tag (2'b01)
…….

            JALR zero, ra, 0      ; check return tag from ra (even check tag of Rloc)
```



Other usage of tagged memory

- Security
  - Stack/heap overflow attack
  - Use after free attack
  - Key protection (prohibit from read)
  - Data flow tracking (tag data sources)
- Performance
  - Identify free space (malloc, garbage collection)

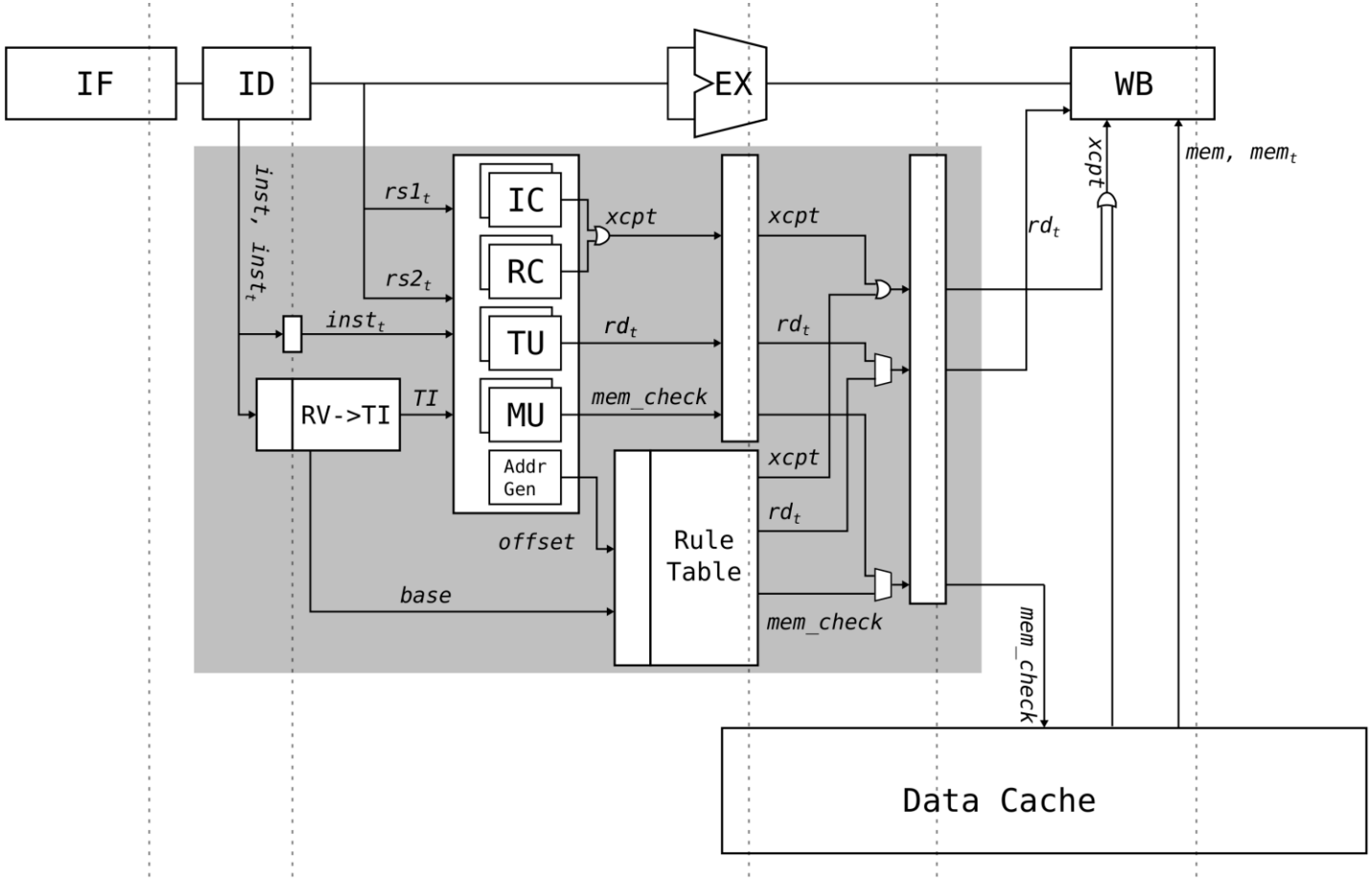
Special Instructions and CSRs

- Tag read and write
  - **TAGR** rd, rs1 ; read the tag of rs1
    (rd_t, rd) <= (0, rs1_t)
  - **TAGW** rd, rs1, imm ; write rd tag to rs1 + imm
    (rd_t, rd) <= (rs1+imm, rd)
- Tag function
  - The tag related logic for tag propagation or check for certain type of instructions (on certain pipeline stage)
  - Every function is controlled by a mask
- 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

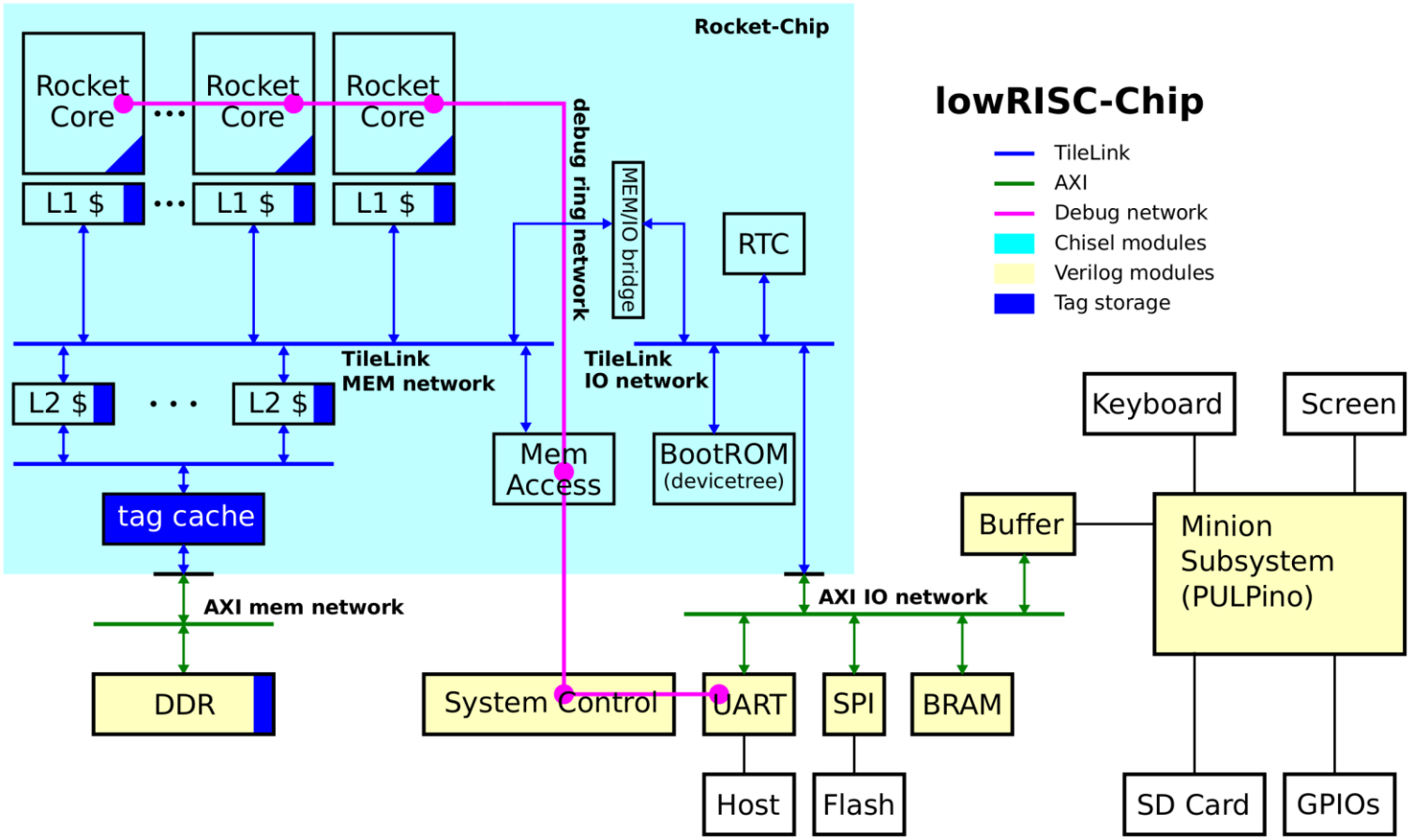
Potential extension

- Dynamic tag rule installation
  - Not only control the mask but also how the mask is interpreted (generic tag function)
  - Different rule sets (use case combination) for different process/thread/library
  - Multiple tag modes in the same system mode (such as U) for different processes/threads (use cases)
  - A tag rule cache addressable by input tags, instruction type and tag mode.
- Off-load tag functions to minions
  - Dynamic rule compiling
  - Dynamic handling of tag exceptions

2-stage tag rule table



Implementation of tagged memory



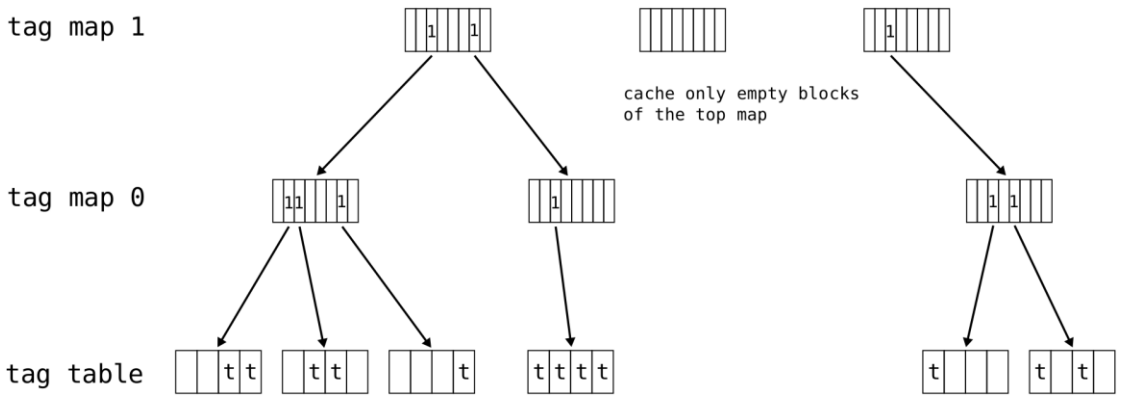
Problems of the Previous Tag Cache

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.

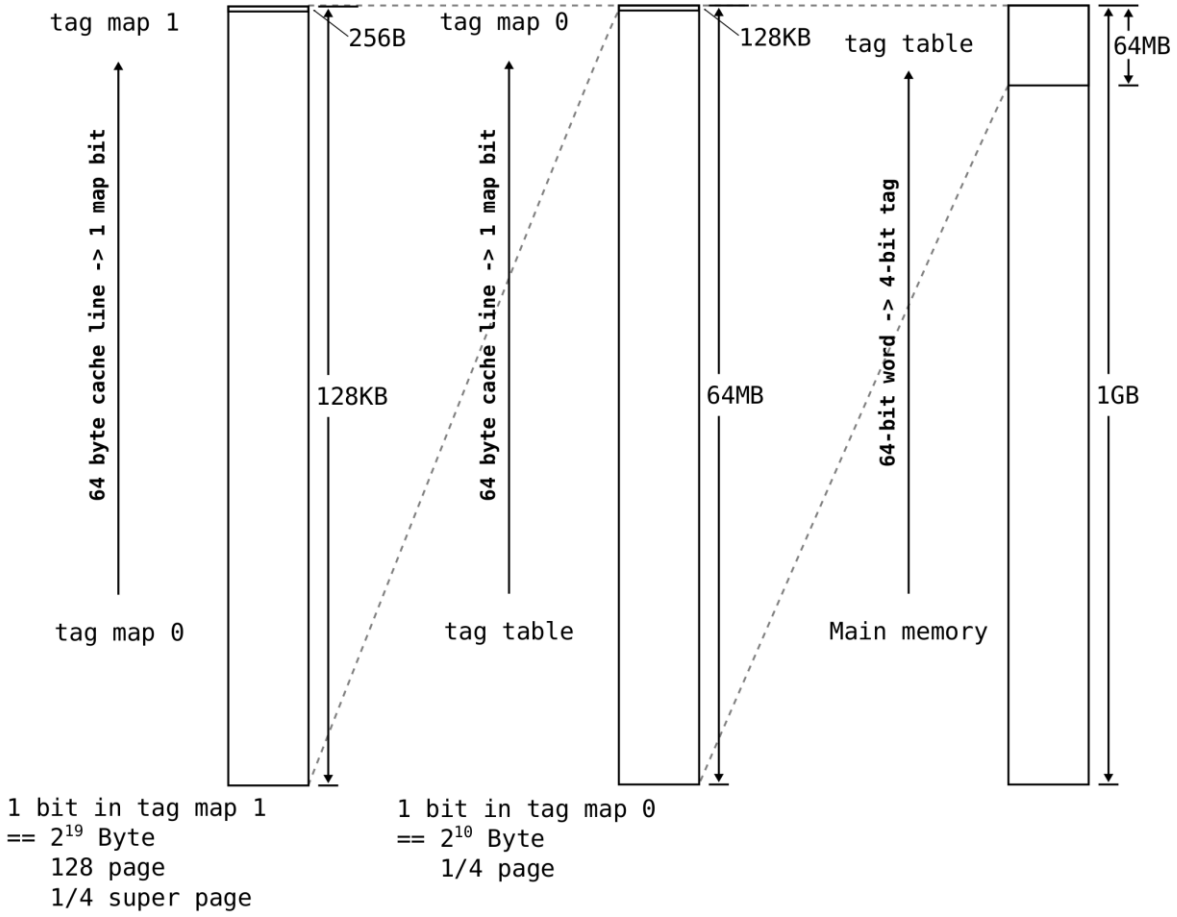
Hierarchical tag cache



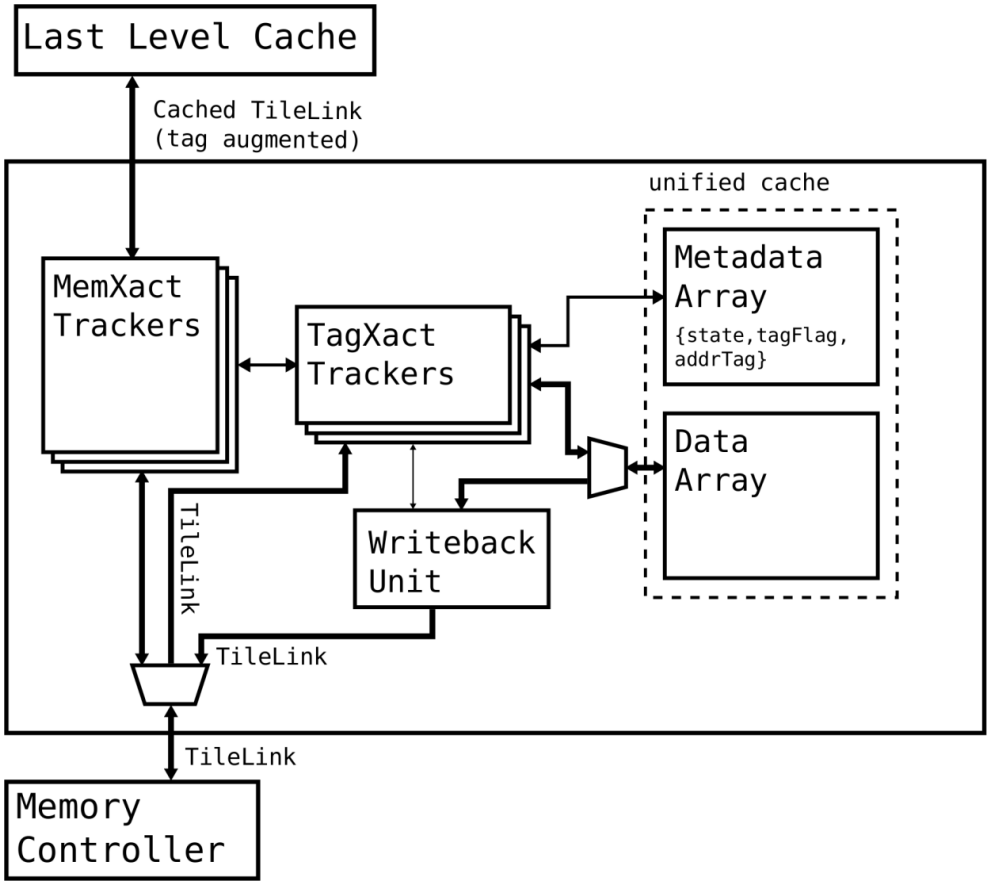
Use a multi-level tag bit-map tree to identify unused cache lines in the tag cache without actually store those lines.

Difficulty: maintaining a consistent tree while support parallel memory accesses.

Hierarchical tag cache



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

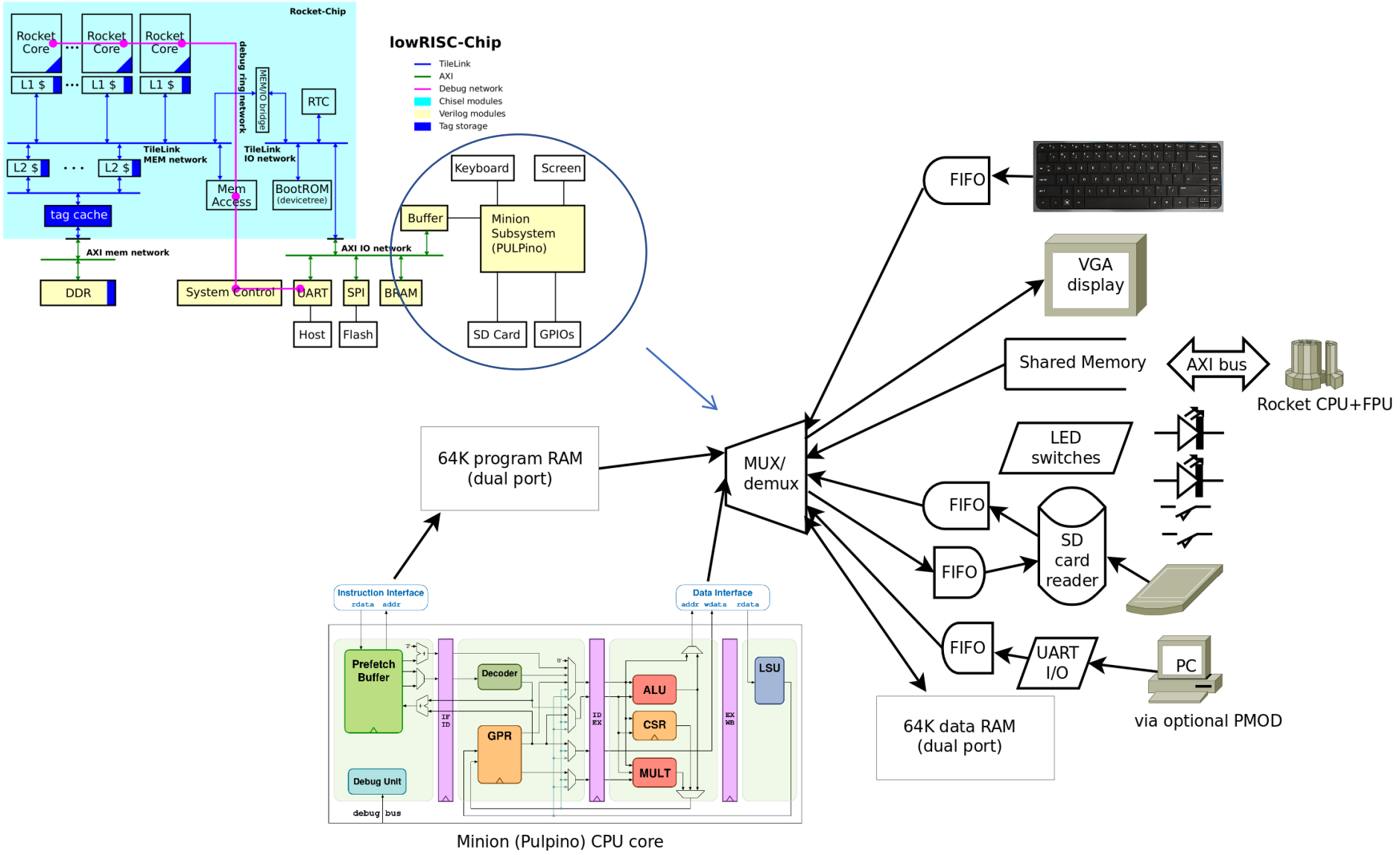
- Maintain consistency between map and table nodes
  - A memory transaction may temporarily break the consistency (non-atomic updating of map bit).
  - An access to the unified tag cache array must be atomic.
  - Block a memory transaction until related map bits return a consistent state.

- 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.

Minion cores

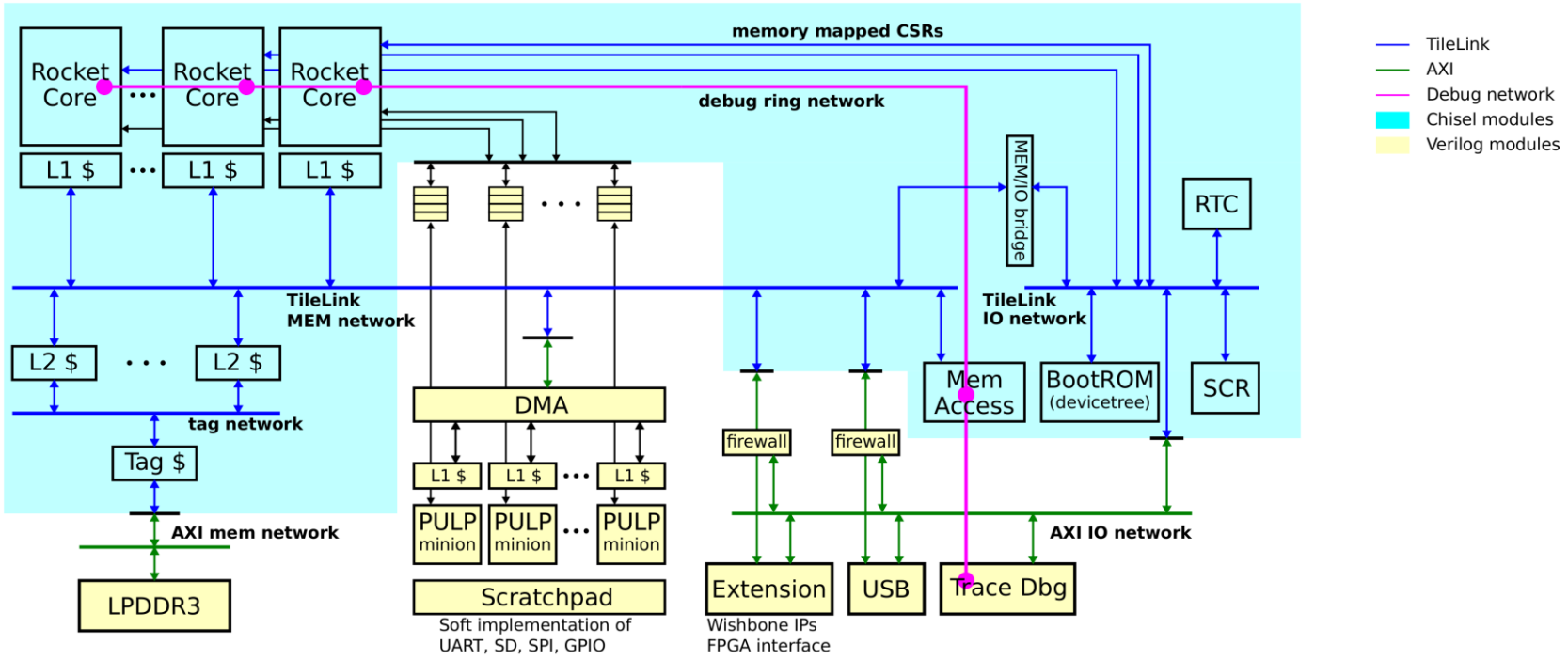
- Motivation
  - The cost of adding transistors is cheap
  - Microprocessors are small
  - Trading area/power with run-time configuration
- I/O processor
  - Programmable device for all low-speed devices
  - A single microprocessor with multiple PHYs
  - Future firmware updates
- Secure boot, hidden security monitor
- Task off-load, accelerator with special ISA extension

Minion System



Minion (Pulpino) CPU core

The expected I/O minions



Trace Debugging

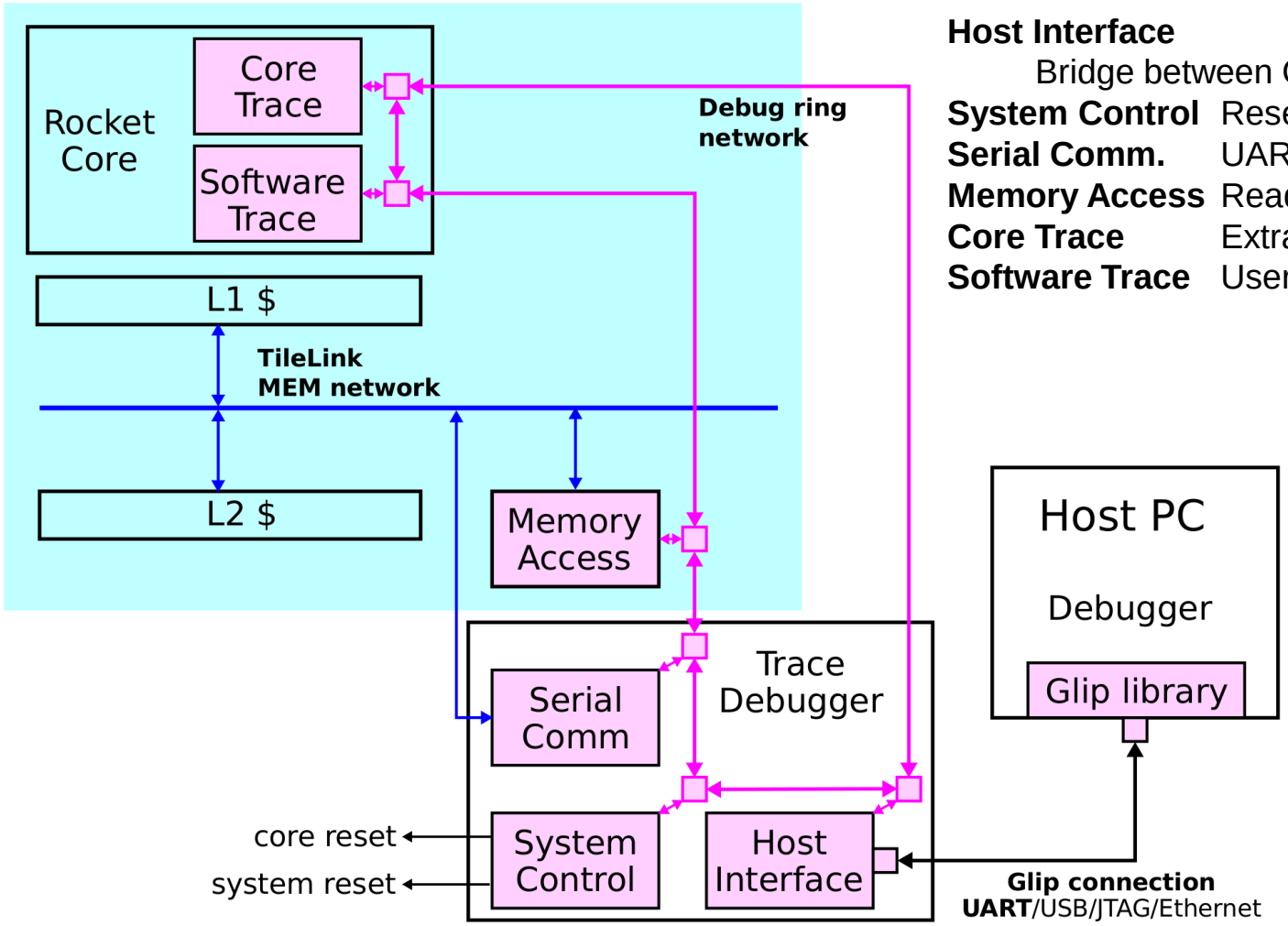
- What is trace debugging
  - Collect instruction and user-defined traces for on/off chip analysis
  - Unlike run-control debugging
  - Non-intrusive, no interruption, minimal performance overhead
- Why use trace debugging
  - Multicore: timing, synchronization, race condition, etc.
  - Detect performance inefficiency
  - Complementary to run-control debugging
  - Light-weight instrumentation

Open SoC Debug

- Open SoC Debug (http://opensocdebug.org)
  - An umbrella project for unified debug infrastructure
  - Provide shared building blocks, interfaces and tools among different platforms

- Design principles
  - **Abstraction from host interface connection**:
    16-bit parallel connection provided by Glip (http://www.glip.io) over UART/USB/JTAG/Ethernet
  - **Easy adoption**: Modular design of debug modules
  - **Unified on-chip communication**: Packet-switched on-chip network connecting all debug components
  - **Functionality**: On-chip trace processing and off-chip trace analyses

Trace Debugger Internals



Host Interface

Bridge between Glip and debug ring

System Control	Reset and pause control
Serial Comm.	UART emulator
Memory Access	Read/write L2
Core Trace	Extract instruction trace
Software Trace	User defined trace

Core Trace

- Function: collect information from the core execution
  - Reconstruct program flow
  - Verify register values
  - Performance analysis
- Trace collection
  - JAL (function call), jump and branch, change of privilege modes
  - ToDo: more traces and run-time configurable filters
- Trace event generation
  - Packetized with timestamp, send to host over debug network
  - Current: Simple overflow handling (drop but record #drops)
  - Future:
    - Better network flow control / QoS
    - Circular buffering and trace recording to DRAM

Example Core Trace

```
# time   event
06570d02 enter init_tls
06570d22 enter memcpy
06570d67 leave memcpy
06570d76 enter memset
06570dae leave memset
06570dcd leave memset
06570dd5 leave init_tls
06570ddb enter thread_entry
06570e22 leave thread_entry
06570e28 enter main
06570e60 enter trace_event0
06570e91 leave trace_event0
06570e96 enter trace_event1
06570ea9 leave trace_event1
06570eb3 enter trace_event2
06570eca leave trace_event2
06570ee3 leave main
06571085 enter exit
065710b3 enter syscall
06571131 change mode to 3
065711ba enter handle_trap
0657127e enter tohost_exit
Overflow, missed 12 events
Overflow, missed 25 events
Overflow, missed 28 events
Overflow, missed 28 events
Overflow, missed 28 events
```

Software Trace

- Function: minimally-invasive code instrumentation
  - Light-weighted alternative to printf()
  - Performance measurement between code points, etc.
  - Can be release unchanged (safety) with minimal performance impact
- Thread-safe trace procedure
  - A trace event: (id, value)
  - Write to $a0 (value), tracked by **Software Trace**
  - Write to a dedicated CSR with (id), which triggers an event
- Trace event generation (same with **Core Trace**)
  - Trace event generation
  - Packetized with timestamp, send to host over debug network
  - Future: Better network flow control / QoS

Example Software Trace

- Trace DMA durations

```
#define TRACE(id,v) \
  asm volatile ("mv   a0,%0": :"r" ((uint64_t)v) : "a0"); \
  asm volatile ("csrw 0x8f0, %0" :: "r"(id));

#define TRACE_DMA_BUFFER(b) TRACE(0x1001,b)
#define TRACE_DMA_START(i,s,b) TRACE(0x1002,i) \
                               TRACE(0x1002,s) \
                               TRACE(0x1002,b)
#define TRACE_DMA_FINISH(i) TRACE(0x1003,i)
```

Header

```
uint8_t *buffer = malloc(42);
TRACE_DMA_BUFFER(buffer);

TRACE_DMA_START(slotid,src,buffer);
dma_transfer(slotid,incoming,buffer);
TRACE_DMA_FINISH(slotid);
```

Source

```
# time   id     value
00002590 0x1001 0xe20c000ac20fc588
00002593 0x1002 0x0000000000000001
00002595 0x1002 0xffff0800000c0000
00002597 0x1002 0xe20c000ac20fc588
00002985 0x1003 0x0000000000000001
```

Trace Log

Visualization

Debug Procedure

Command Line Interface

```
# reset and pause cores
reset -halt
# load a test program
mem loadelf test.elf 3
# enable core trace
ctm log ctm.log 4
# enable software trace
stm log stm.log 5
# open a terminal (xterm)
terminal 2
# run the test
start
```

ID	*Module*
0	Host interface
1	System Control
2	Ser. Comm.
3	Mem. Access
4	Core Trace
5	Software Trace

Python Script

```
import opensocdebug
import sys

if len(sys.argv) < 2:
    print "Usage: runelf.py <filename>"
    exit(1)

elffile = sys.argv[1]

osd = opensocdebug.Session()

osd.reset(halt=True)

for m in osd.get_modules("STM"):
    m.log("stm{:03x}.log".format(m.get_id()))

for m in osd.get_modules("CTM"):
    m.log("ctm{:03x}.log".format(m.get_id()), elffile)

for m in osd.get_modules("MAM"):
    m.loadelf(elffile)

osd.start()
```

Future work of lowRISC

- Rocket
  - Merge the upstream code from SiFive/UC Berkeley
    - Latest Rocket with priv 1.10 (L2 $?)
    - Tilelink2
    - Chisel3
  - Toolchain
    - LLVM
    - Tag support in GCC/LLVM
- Tagged memory
  - Benchmark for current implementation
  - Tag expansion for generic tag policies
- Minion
  - More structural work
- Trace debugger
  - Merge with run-control debugger (GDB)
  - More features in trace filters and triggers

Get Involved

- lowRISC is an opensourced and free SoC provider that produces Linux capable multicore SoC platforms.
  - All on GitHub, no hidden code.
  - Submit pull request for bug fixes.
  - Contact us for ideas, improvement, extensions.

Contribution is needed …

Peripherals, testing, compiler, Linux kernel, benchmarking, etc.

Website:	http://www.lowrisc.org/
Mail list:	lowrisc-dev@lists.lowrisc.org
GitHub:	https://github.com/lowrisc/
E-mail:	info@lowrisc.org