EECS251B : Advanced Digital Circuits and Systems

Lecture 7 – RoCC and TileLink

Borivoje Nikolić
Announcements

• Lab 3 due this week
• Homework 2 will be posted at the end of the week
• Project teaming this week
• Accelerator Integration
• Tightly-coupled Acc. w/ RoCC
• MMIO Acc. w/ TileLink
• Examples
Domain-Specific Accelerators

- Customized hardware designed for a domain of applications.

Apple M1 Chip 2020

* AnandTech
Accelerators don’t exist in isolation.

http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/
Mobile SoC Usecases

- Mainstream architecture has long focused on general-purpose CPUs and GPUs.
- In an SoC, multiple IP blocks are active at the same time and communicate frequently with each other.
- Example:
  - Recording a 4K video
  - Camera -> ISP
    - “Preview stream” for display
    - “Video stream” for storage
  - DRAM for data sharing
Mobile SoC Usecases

- Multiple accelerators are running concurrently for different usecases.

<table>
<thead>
<tr>
<th>Accelerators (IPs)</th>
<th>CPUs (AP)</th>
<th>Display</th>
<th>Media Scaler</th>
<th>GPU</th>
<th>Image Signal Proc.</th>
<th>JPEG</th>
<th>Pixel Visual Core</th>
<th>Video Decoder</th>
<th>Video Encoder</th>
<th>Dozens More</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photo Enhancing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video Capture</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Video Capture HDR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Video Playback</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Image Recognition</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mark Hill and Vijay Janapa Reddi, *Gables: A Roofline Model for Mobile SoCs*, HPCA’2019
Full-System Visibility for DL Accelerators

SoC

Gemmini 1
- Spatial Array
- Scratches

CPU 1
- RF
- TLB
- Private L1$

CPU 2
- RF
- TLB
- Private L1$

Gemmini 2
- Spatial Array

Shared L2 Cache

IOs, Interconnects, etc.
Full-System Visibility: Memory Hierarchy

Performance Impacts
Resource contention, cache coherence, etc.
Full-System Visibility: Virtual Addresses

Performance Impacts
Page faults, TLB hits, etc.
Full-System Visibility: Host CPUs

Performance Impacts
Unaccelerated kernels, etc.
• Accelerator Integration
• Tightly-coupled Acc. w/ RoCC
• MMIO Acc. w/ TileLink
• Examples
Rocket Custom Coprocessor Interface (RoCC)

- An interface to facilitate easy decoupled communications between the core and the attached coprocessors.
- The RoCC interface accepts coprocessor commands generated by the Rocket core.
RoCC Instruction Format

- The commands include the instruction word and the values in up to two integer registers, and commands may write an integer register in response.

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>xd</th>
<th>xs1</th>
<th>xs2</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
</table>

- Xd, xs1, and xs2 are used as valid bits for the register specifiers as whether the core is using those registers.

<table>
<thead>
<tr>
<th>funct7</th>
<th>xd</th>
<th>xs1</th>
<th>xs2</th>
<th>inst_rd</th>
<th>inst_rs1</th>
<th>inst_rs2</th>
<th>rs1</th>
<th>rs2</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Core</td>
<td>Acc</td>
<td>-</td>
<td>data1</td>
<td>-</td>
<td>C[inst_rd] ← A[inst_rs1]</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Core</td>
<td>Core</td>
<td>-</td>
<td>data1</td>
<td>-</td>
<td>C[inst_rd] ← A[data1]</td>
</tr>
<tr>
<td>write</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Acc</td>
<td>Core</td>
<td>-</td>
<td>data1</td>
<td>-</td>
<td>A[inst_rd] ← data1</td>
</tr>
<tr>
<td>write</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Acc</td>
<td>Core</td>
<td>Core</td>
<td>data1</td>
<td>data2</td>
<td>A[data2] ← data1</td>
</tr>
<tr>
<td>load</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Acc</td>
<td>-</td>
<td>Core</td>
<td>data1</td>
<td>-</td>
<td>A[inst_rd] ← M[data1]</td>
</tr>
<tr>
<td>load</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>Core</td>
<td>Core</td>
<td>data1</td>
<td>data2</td>
<td>A[data2] ← M[data1]</td>
</tr>
<tr>
<td>store</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>Core</td>
<td>Acc</td>
<td>data1</td>
<td>-</td>
<td>M[data1] ← A[inst_rs2]</td>
</tr>
<tr>
<td>store</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>Core</td>
<td>Core</td>
<td>data1</td>
<td>data2</td>
<td>M[data1] ← A[data2]</td>
</tr>
</tbody>
</table>
Extended RoCC Interface

• Allows the attached coprocessor to share the Rocket core’s data cache and page table walker and provides a facility for the coprocessor to interrupt the core.
  • These mechanisms are sufficient to construct coprocessors that participate in a page-based virtual memory system.

• RoCC accelerators may connect to the outer memory system directly over the TileLink interconnect, providing a high-bandwidth but coherent memory port.
• Accelerator Integration
• Tightly-coupled Acc. w/ RoCC
• MMIO Acc. w/ TileLink
• Examples
Memory-Mapped IO Accelerators

- Loosely-coupled accelerators
  - Communicates with the core through memory-mapped registers.
  - Instead of being invoked directly through RoCC instructions.

- A commonly-used way to connect loosely-coupled accelerators on an SoC.
  - Access shared data in LLC and/or DRAM
  - Can be coherent or not
  - ARM’s AXI
  - RISC-V’s TileLink

https://developer.arm.com/documentation/102202/0200/What-is-AMBA--and-why-use-it-
TileLink

- A chip-scale interconnect standard providing coherent memory-mapped access to memory and other devices.
- Designed for use in a system-on-chip (SoC) to connect general-purpose multiprocessors, co-processors, accelerators, DMA engines.
- Free and open-source
- RISC-V-based systems
TileLink Protocol Levels

- TileLink Uncached Lightweight (TL-UL)
  - Only simple memory read/write (Get/Put) operations of single words (similar to AXILite)

- TileLink Uncached Heavyweight (TL-UH)
  - Adds various hints, atomic, and burst accesses but w/o coherence (similar to AXI4)

- TileLink Cached (TL-C)
  - Complete protocol, which supports use of coherent caches (similar to ACE)

<table>
<thead>
<tr>
<th></th>
<th>TL-UL</th>
<th>TL-UH</th>
<th>TL-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Write Operations</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Multibeat Messages</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Atomic Operations</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Hint (Prefetch) Operations</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Cache Block Transfers</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Priorities B+C+E</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

## TileLink Channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Direction</th>
<th>Purpose</th>
<th>(TL-C only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Manager to Subordinate</td>
<td>Request messages sent to an address</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Subordinate to Manager</td>
<td>Request messages sent to a cached block</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Manager to Subordinate</td>
<td>Response messages from a cached block</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Subordinate to Manager</td>
<td>Response messages from an address</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Manager to Subordinate</td>
<td>Final handshake for cache block transfer</td>
<td></td>
</tr>
</tbody>
</table>

* Using AXI agent names here
* TileLink Client -> AXI Manager
* TileLink Manager -> AXI Subordinate
TileLink Basics: Messages

- Messages composed of Beats (one beat per clock cycle) containing:
  - The unchanging message header, including
    - opcode: the message type
    - size: base-2 log of the number of bytes in the data payload
  - Multi-Beat data payload
    - Number of Beats calculated from the message size
    - A burst is said to be in progress after the first beat has been accepted and until the last beat has been accepted.
    - When a burst is in progress, if valid is HIGH, the sender must additionally present:
      - Only a beat from the same message burst.
      - Control signals identical to those of the first beat.
      - Data signals corresponding to the previous beat’s address plus the data bus width in bytes.
      - Final signals changing only on the final beat.
TileLink Basics: Flow-control

• Beats are regulated by **ready-valid** handshake

• The receiver provides **ready**
  • If ready is LOW, the receiver must not process the beat and the sender must not consider the beat processed

• The sender provides **valid** + the beat payload
  • If valid is LOW, the payload may be an illegal TileLink message
  • valid must never depend on ready
    • If a sender wishes to send a Beat, it must assert valid independently of whether the receiver signals that it is ready.

• Avoiding deadlock
  • Rules that govern the conditions under which a receiving agent may reject a beat of a message by lowering ready.
  • Rules on allowable topologies of a TileLink network: The structure of agents and links must be a Directed Acyclic Graph (DAG).
A beat is exchanged only when both ready and valid are HIGH
TileLink Basics: Request-Response

Max and min delay between a Get (4) and an AccessAckData (1) on an 8-byte bus.

Max and min delay between a PutFullData (0) and an AccessAck (0) on an 8-byte bus

- Response as early as the same cycle
- Timeouts are forbidden
TileLink: The Foundation of SiFive’s FU500

TileLink Examples

• RoCC accelerators: SHA3
  • https://github.com/ucb-bar/sha3

• RoCC + TL-UL: protobuf accelerator
  • https://github.com/ucb-bar/protoacc

• RoCC + TL-UH: Gemmini accelerator
  • https://github.com/ucb-bar/gemmini

• RoCC + TL-UH: Hwacha vector accelerator
  • https://github.com/ucb-bar/hwacha

• TL-UH: IceNIC network interface controller for FireSim
  • https://github.com/firesim/icenet
Instantiate a TileLink node for your module

```scala
344  class StreamWriter[T <: Data: Arithmetic](nXacts: Int, beatBits: Int, maxBytes: Int, dataWidth: Int, aligned_to: Int,
345     inputType: T, block_cols: Int, use_tlb_register_filter: Boolean,
346     use_firesim_simulation_counters: Boolean)
347     (implicit p: Parameters) extends LazyModule {
348       val node = TLHelper.makeClientNode(
349         name = "stream-writer", sourceId = IdRange(0, nXacts))
350
351       require(isPow2(aligned_to))
352
353       lazy val module = new LazyModuleImp(this) with HasCoreParameters with MemoryOpConstants {
354         val (tl, edge) = node.out(0)
355         val dataBytes = dataWidth / 8
356         val beatBytes = beatBits / 8
357         val lgBeatBytes = log2Ceil(beatBytes)
358         val maxBeatsPerReq = maxBytes / beatBytes
359         val inputTypeRowBytes = block_cols * inputType.getPitch / 8
360         val maxBlocks = maxBytes / inputTypeRowBytes
```

• Accelerator Integration
• Tightly-coupled Acc. w/ RoCC
• MMIO Acc. w/ TileLink
• Examples
Gemmini: Full-System Co-Design of Hardware Accelerators

- Full-stack
  - Includes OS
  - End-to-end workloads
  - “Multi-level” API

- Full-SoC
  - Host CPUs
  - Shared memory hierarchies
  - Virtual address translation

---

<table>
<thead>
<tr>
<th>Property</th>
<th>NVDLA</th>
<th>VTA</th>
<th>PolySA</th>
<th>DNNBuilder</th>
<th>MAGNet</th>
<th>DNNWeaver</th>
<th>MAERI</th>
<th>Gemmini</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Architecture Template</td>
<td>Multiple Datatypes Multiple Dataflows Spatial Array Direct convolution</td>
<td>Int/Float</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Int</td>
<td>Int/Float</td>
</tr>
<tr>
<td>Programming Support</td>
<td>Software Ecosystem Hardware-Supported Virtual Memory</td>
<td>Custom Compiler</td>
<td>TVM</td>
<td>Xilinx SDAccel</td>
<td>Caffe</td>
<td>Caffe</td>
<td>Custom Mapper</td>
<td>ONNX/C</td>
</tr>
<tr>
<td>System Support</td>
<td>Full SoC OS Support</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://github.com/ucb-bar/gemmini

[2021 Best Paper Award]
Using RoCC + TileLink w/ Gemmini

- How **Gemmini**, a DNN accelerator, uses RoCC and TileLink
- How does Gemmini **read** data from main memory into Gemmini’s scratchpad?

1. Host CPU encounters **unknown** RISC-V instruction

<table>
<thead>
<tr>
<th>Unknown opcode</th>
<th>rs1</th>
<th>rs2</th>
</tr>
</thead>
</table>
Using RoCC + TileLink w/ Gemmini

• How **Gemmini**, a DNN accelerator, uses RoCC and TileLink
  • How does Gemmini **read** data from main memory into Gemmini’s scratchpad?

1. Host CPU encounters **unknown** RISC-V instruction

2. Host CPU **dispatches** unknown instruction to RoCC accelerator
   a. As well as Rs1 and Rs2 contents (128 bits extra bits)
Using RoCC + TileLink w/ Gemmini

• How Gemmini, a DNN accelerator, uses RoCC and TileLink
  • How does Gemmini read data from main memory into Gemmini’s scratchpad?

1. Host CPU encounters unknown RISC-V instruction
2. Host CPU dispatches unknown instruction to RoCC accelerator
   a. As well as Rs1 and Rs2 contents (128 bits extra bits)
3. Gemmini decodes instruction
   a. It’s a load instruction!

<table>
<thead>
<tr>
<th>Unknown opcode</th>
<th>rs1</th>
<th>rs2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Load data</th>
<th>Main memory address</th>
<th>Scratchpad address</th>
</tr>
</thead>
</table>
Using RoCC + TileLink with Gemmini

• How Gemmini, a DNN accelerator, uses RoCC and TileLink
  • How does Gemmini read data from main memory into Gemmini’s scratchpad?

1. Host CPU encounters **unknown** RISC-V instruction

2. Host CPU **dispatches** unknown instruction to RoCC accelerator
   a. As well as Rs1 and Rs2 contents (128 bits extra bits)

3. Gemmini **decodes** instruction
   a. It’s a load instruction!

4. Gemmini asks CPU’s page table walker to **translate** addresses in Rs1, Rs2
   a. PTW is only available through RoCC interface
Using RoCC + TileLink w/ Gemmini

• How **Gemmini**, a DNN accelerator, uses RoCC and TileLink
  • How does Gemmini read data from main memory into Gemmini’s scratchpad?

1. Host CPU encounters **unknown** RISC-V instruction

2. Host CPU **dispatches** unknown instruction to RoCC accelerator
   a. As well as Rs1 and Rs2 contents (128 bits extra bits)

3. Gemmini **decodes** instruction
   a. It’s a load instruction!

4. Gemmini asks CPU’s page table walker to **translate** addresses in Rs1, Rs2
   a. PTW is only available through RoCC interface

5. Gemmini sends **TileLink requests** to read data from main memory
   a. Often, multiple TileLink requests must be sent, due to TileLink’s alignment and length limitations
Review

- Accelerators don’t exist in isolation.
- RoCC for tightly-coupled accelerators
- TileLink for loosely-coupled, MMIO accelerators

Examples:
- RoCC accelerators: SHA3
  - [https://github.com/ucb-bar/sha3](https://github.com/ucb-bar/sha3)
- RoCC + TL-UL: protobuf accelerator
  - [https://github.com/ucb-bar/protoacc](https://github.com/ucb-bar/protoacc)
- RoCC + TL-UH: Gemmini accelerator
  - [https://github.com/ucb-bar/gemmini](https://github.com/ucb-bar/gemmini)
- RoCC + TL-UH: Hwacha vector accelerator
  - [https://github.com/ucb-bar/hwacha](https://github.com/ucb-bar/hwacha)
- TL-UH: IceNIC network interface controller for FireSim
  - [https://github.com/firesim/icenet](https://github.com/firesim/icenet)