

CORE-ET CPU Subsystem Micro- Architecture Reference

Table of Contents

Document Control	1
Revision History	1
1. ET-SoC-1 Overview	2
1.1. ET-SoC-1 at a Glance	2
1.2. The ET-Minion Hierarchy	2
1.2.1. ET-Minion Core	3
1.2.2. Neighborhood	4
1.2.3. Minion Shire	5
1.2.4. Minion Shire Array	6
1.3. ET-Maxion and Service Processor	7
1.3.1. ET-Maxion Cores	7
1.3.2. Service Processor	7
1.4. Mesh Network-on-Chip	8
1.4.1. Mesh Topology and Mesh Stops	8
1.4.2. Memory Shires	9
1.4.3. PCIe Shire	9
1.4.4. I/O Shire	9
1.5. ET-SoC-1 Accelerator Card	9
2. ET-Minion Core	11
2.1. Core Overview	11
2.1.1. Block Partitioning	11
2.1.2. Hardware Threads	12
2.1.3. Instruction Flow Through the Core	12
2.1.4. Interfaces to the Neighborhood	13
2.1.5. Clocks, Resets and Power	13
2.1.6. Debug Support	13
2.2. Front End	13
2.2.1. Overview	13
2.2.2. Interface	15
2.2.3. Pipeline Stages	20
2.2.4. Interaction with the L0 Micro-Cache	21
2.2.5. Thread Buffer	25
2.2.6. Entry Invalidation, Redirect and Thread Start	29
2.2.7. Decoders	31
2.2.8. Instruction FIFO	32
2.2.9. Thread Scheduler	33
2.2.10. Halt and Debug Program Buffer	34
2.3. Integer Pipeline	36

2.3.1. Overview	36
2.3.2. Pipeline Stages	36
2.3.3. Interfaces	36
2.3.4. Register File and Operand Read	36
2.3.5. Immediate Generation and ALU	36
2.3.6. Multiply/Divide Unit	36
2.3.7. Scoreboards and Hazards	36
2.3.8. Hart Interleaving and Stalls	36
2.3.9. Branches, Redirects and Replay	36
2.3.10. Interface to the Data Cache	36
2.3.11. Interface to the VPU	36
2.3.12. CSRs and Privilege Modes	36
2.3.13. Exceptions and Interrupts	36
2.3.14. Debug and Breakpoints	36
2.3.15. Performance Counters	37
2.3.16. Verification Notes	37
2.4. Data Cache	37
2.4.1. Overview	37
2.4.2. Organization and Operating Modes	37
2.4.3. Interfaces	37
2.4.4. Request Pipeline	37
2.4.5. Address Translation and PMA Checks	37
2.4.6. Tag, Data and LRU Arrays	37
2.4.7. Miss Handling	37
2.4.8. Replay Queue	37
2.4.9. Writeback and Store Merge	37
2.4.10. Atomic Operations	37
2.4.11. Cache-Management Operations	37
2.4.12. Scratchpad Control	37
2.4.13. Tensor Load and Reduce	37
2.4.14. Interface to the Neighborhood	37
2.4.15. Errors	37
2.4.16. Performance	37
2.4.17. Verification Notes	38
2.5. Vector Processing Unit	38
2.5.1. Overview	38
2.5.2. Interfaces	38
2.5.3. Control and Micro-Instruction Decode	38
2.5.4. Register Files	38
2.5.5. Lanes	38
2.5.6. Mask Unit	38

2.5.7. Bypass Network	38
2.5.8. Shift and Swizzle Unit	38
2.5.9. TxFMA Unit	38
2.5.10. Integer Multiply-Accumulate Unit	38
2.5.11. Transcendental Unit	38
2.5.12. Tensor Operations	38
2.5.13. Interfaces to the Integer Pipeline and Data Cache	38
2.5.14. Performance	38
2.5.15. Verification Notes	38
3. Neighborhood	39
3.1. Neighborhood Overview	39
3.1.1. Partitioning	39
3.1.2. Shared Resources and Core Connections	39
3.1.3. Voltage-Domain Crossings	39
3.1.4. External Interfaces	39
3.2. Front End – I-Cache Interface	39
3.2.1. Overview	39
3.2.2. Interface Signals	39
3.2.3. Request Arbitration	39
3.2.4. Request and Response Timing	39
3.2.5. Miss, Sleep and Fill-Done Protocol	39
3.2.6. Fault and Error Reporting	39
3.2.7. Livelock Avoidance	39
3.2.8. Known Limitations	39
3.3. L0 Micro-Cache	39
3.3.1. Overview	40
3.3.2. Internal Blocks	40
3.3.3. Pipeline Stages	40
3.3.4. Instruction TLB	40
3.3.5. Micro-Tag and Micro-Data Arrays	40
3.3.6. PMA Check	40
3.3.7. Replacement	40
3.3.8. Miss Handling	40
3.3.9. L1 Bypass	40
3.3.10. Software Prefetch	40
3.4. L1 Instruction Cache	40
3.4.1. Overview	40
3.4.2. Organization	40
3.4.3. Tag, LRU and Data Arrays	40
3.4.4. External Data RAM	40
3.4.5. Request Arbitration Between L0 Micro-Caches	40

3.4.6. Miss Path	41
3.4.7. Errors and ECC	41
3.5. Shared Page-Table Walker	41
3.5.1. Clients and Arbitration	41
3.5.2. Walk Sequence and Page Sizes	41
3.5.3. PTE Cache and Invalidation	41
3.6. Memory Request and Response Path	41
3.6.1. Miss, Evict and Fill FIFOs	41
3.6.2. Request Arbitration	41
3.6.3. Response Routing and Backpressure	41
3.7. Tensor Support	41
3.7.1. Cooperative Tensor Load	41
3.7.2. Tensor Store Buffer	41
3.7.3. TBOX Router	41
3.8. Synchronization, Messaging, Registers and PMU	41
3.8.1. Fast Local Barrier	41
3.8.2. Local Message Network	41
3.8.3. IPI and FCC Paths	41
3.8.4. Neighborhood Registers and APB Access	41
3.8.5. Debug Channel	41
3.8.6. Performance Monitoring Unit	42

Document Control

Revision History

Version	Date	Author	Notes
v0.1	2026-09-22	Ryan Naveed	• First draft of the document. • Chapter 1, ET-SoC-1 Overview: written. • Chapter 2, ET-Minion Core: Core Overview (block partitioning, hardware threads, instruction flow) and Front End written. Integer Pipeline, Data Cache and VPU contain headings only. • Chapter 3, Neighborhood: headings only.

1. ET-SoC-1 Overview

1.1. ET-SoC-1 at a Glance

ET-SoC-1 is a many-core RISC-V system-on-chip developed primarily for artificial-intelligence and machine-learning (AI/ML) workloads. It is designed as a flexible, fast and efficient inference engine for ML applications. Its compute resources are arranged in a multi-level hierarchy that delivers high sustained performance on regular, parallelizable workloads, such as those of ML algorithms.

The device integrates 1,093 general-purpose 64-bit RISC-V cores of three types:

- 1,088 ET-Minion cores for ML computation
- 4 ET-Maxion cores for management and general-purpose tasks
- 1 Service Processor for boot and device management

The cores are connected to on-die SRAM, external memory and a set of standard interfaces through a mesh network-on-chip (NoC). [Table 1](#) summarizes the device.

Table 1. ET-SoC-1 key characteristics

Item	Description
ET-Minion cores	1,088 in-order, dual-threaded RV64IMFC cores, each with an 8-lane vector/tensor unit
ET-Maxion cores	4 single-threaded, superscalar, out-of-order 64-bit RISC-V cores
Service Processor	1 single-threaded core derived from the ET-Minion
On-die SRAM	Each Minion Shire's 4 MB Shire Cache can be partitioned by ESRs into L2, a slice of the chip-wide L3, and scratchpad.
Interconnect	Mesh NoC with 44 mesh stops
External memory	LPDDR4X, 16 channels x 16 bits, up to 4,266 MT/s
PCI Express	Gen4, 8 lanes. Endpoint, root complex, or both at once.
Other interfaces	eMMC, USB 2.0 OTG, I2C/SMBus, I3C, SPI, UART, GPIO. Debug access through JTAG and a USB 2.0 device port.

1.2. The ET-Minion Hierarchy

The ET-Minion cores are organized in three levels, as shown in [Figure 1](#):

- **ET-Minion core**
- **Neighborhood**: 8 ET-Minion cores
- **Minion Shire**: 4 Neighborhoods (32 ET-Minion cores)

Each level adds resources that are shared by the units it contains. The device contains 34 Minion Shires, which together form the Minion Shire array ([Section 1.2.4](#)).

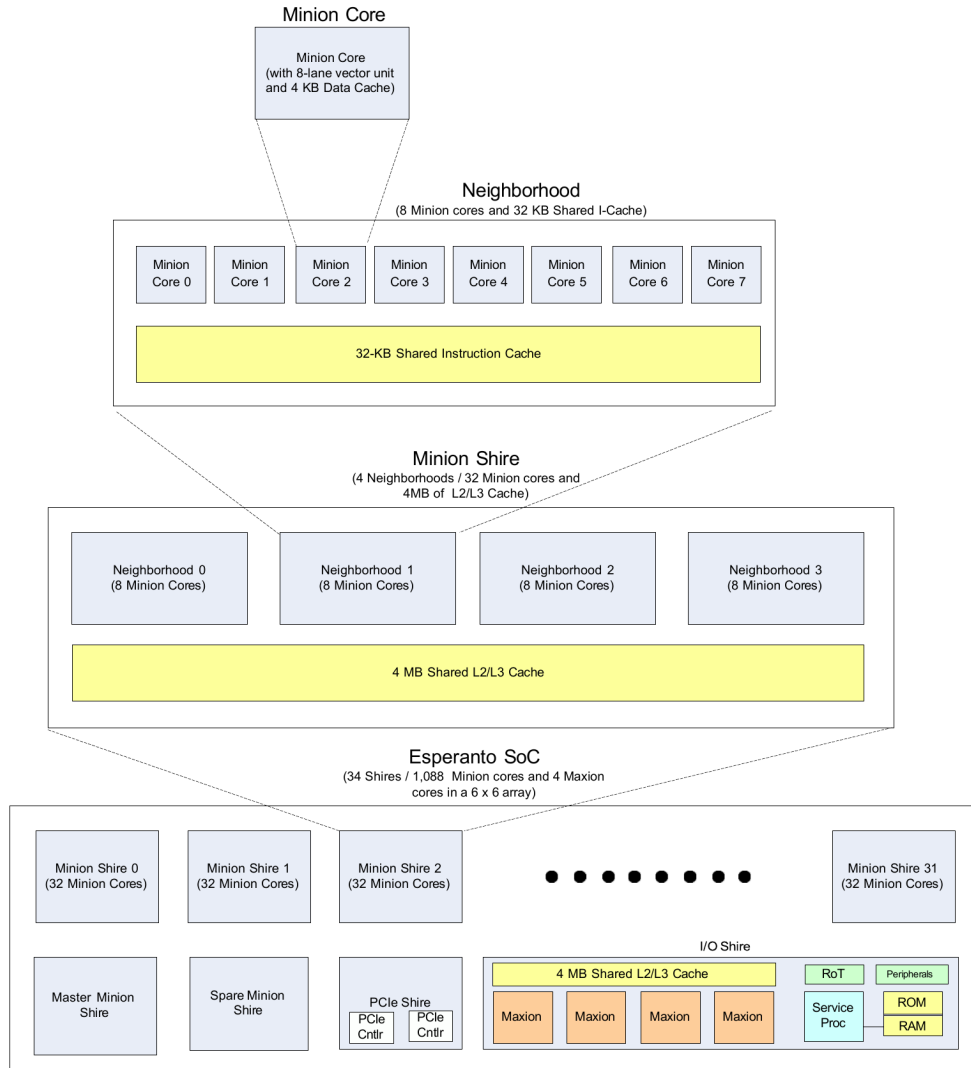


Figure 1. ET-Minion hierarchy

1.2.1. ET-Minion Core

The ET-Minion is a dual-threaded, in-order, single-issue 64-bit RISC-V core that implements RV64IMFC. Its main features are:

- Two hardware threads (harts)
- 4 KB private L1 data cache, partly configurable as scratchpad memory
- Vector processing unit (VPU) with eight identical lanes operating in lock-step. Each lane contains:
 - Floating-point multiply-add unit (FMA): one 32-bit or two 16-bit operations
 - Two integer multiply-add units (IMA): four 8-bit multiply-accumulates each
 - Integer unit (INT)
 - Transcendental unit (TRANS)
- Esperanto instruction extensions for vector, tensor, atomic, message, cache-management and fast-synchronization operations

The core consists of four top-level blocks, shown in [Figure 2](#):

- **Front end (FE)**
- **Integer pipeline**
- **L1 data cache and scratchpad**
- **VPU**

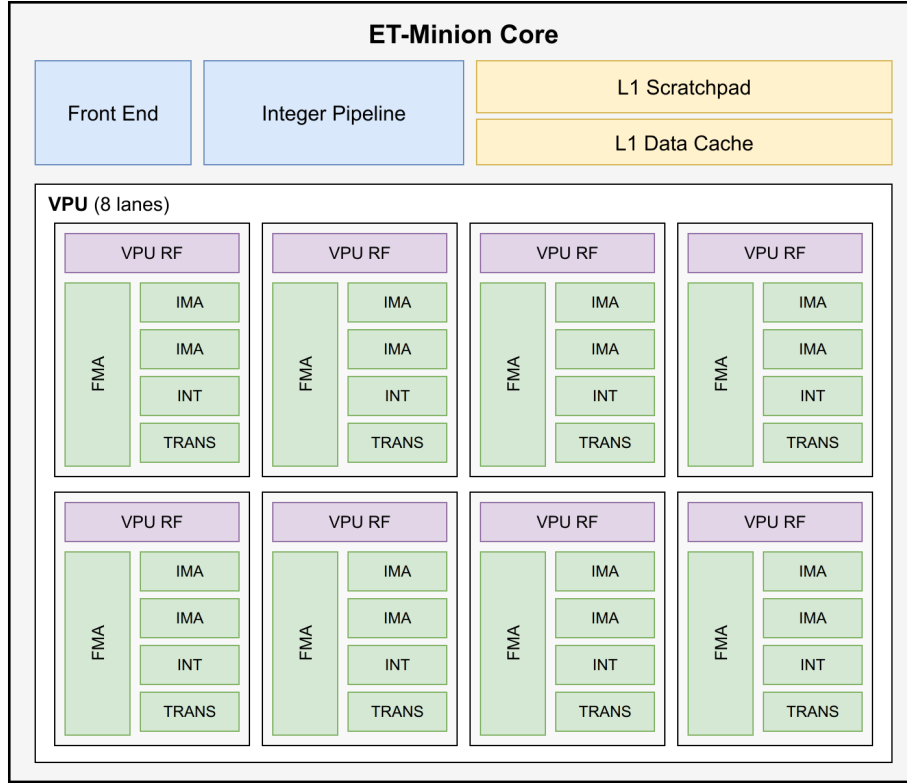


Figure 2. ET-Minion core

1.2.2. Neighborhood

A Neighborhood groups eight ET-Minion cores that share the following resources. In Figure 3, the L0 micro-caches are labeled "I-Cache".

- **L1 instruction cache:** 32 KB, organized as 128 sets $\times$ 4 ways with 64-byte lines. It is shared by all eight cores, and its data RAMs are located outside the Neighborhood.
- **L0 micro-caches:** two, each shared by four cores. Each holds 16 fully associative cache lines, sits between its cores and the L1 instruction cache, and accesses the L1 instruction cache on a miss.
- **Page-table walkers (PTWs):** two, each shared by four cores. Each PTW serves the data caches of its four cores and the instruction-cache side.
- **Request/response path:** a single path to the memory hierarchy outside the Neighborhood. Requests from the eight data caches, the instruction cache, the two PTWs and, the cooperative TensorLoad/TensorStore units are arbitrated into (256-bit internally which is converted to a 512-bit ET-Link bus).
- **Performance monitoring unit (PMU):** twelve 64-bit event counters shared by the eight cores. Eight counters count ET-Minion events and four count Neighborhood events.
- **Cooperative TensorLoad:** coalesces the TensorLoad requests that several cores issue for the same address into a single memory request, and delivers the one response back to every

cooperating core. Cores in all four Neighborhoods of a Minion Shire can cooperate; one Neighborhood acts as master and sends the combined request.

- **Cooperative TensorStore:** combines the partial writes of cooperating cores into one Shire Cache write, and replicates the acknowledgement to every core that took part. Cores cooperate in two fixed groups, cores 0 to 3 and cores 4 to 7, in one of three modes:
 - **Quad-128:** 128 bits from each of four cores, forming a full 512-bit cache line
 - **Pair-256:** 256 bits from each of two cores, forming a full 512-bit cache line
 - **Pair-128:** 128 bits from each of two cores, forming half a cache line

Each core holds its request until every core of the group has presented one, so operations of different modes cannot mix.

- **Fast Local Messaging Network (FLN):** carries messages between cores of the same Neighborhood directly, instead of sending them out through the request path and back in through the response path. Connections are not all-to-all; they follow the pattern used by the tensor reduction instructions, and a message arriving over this network takes priority over the other responses to that core.
- **Fast local barrier (FLB) path**
- **Neighborhood ESRs**
- **Interrupts:** routes the external, timer and software interrupt lines to the cores, and delivers the inter-processor interrupts (IPIs). A software write to a Shire ESR raises a machine software interrupt for a hart, or a redirect IPI that sends the selected harts to the PC held in the Neighborhood `ipi_redirect_pc` ESR.

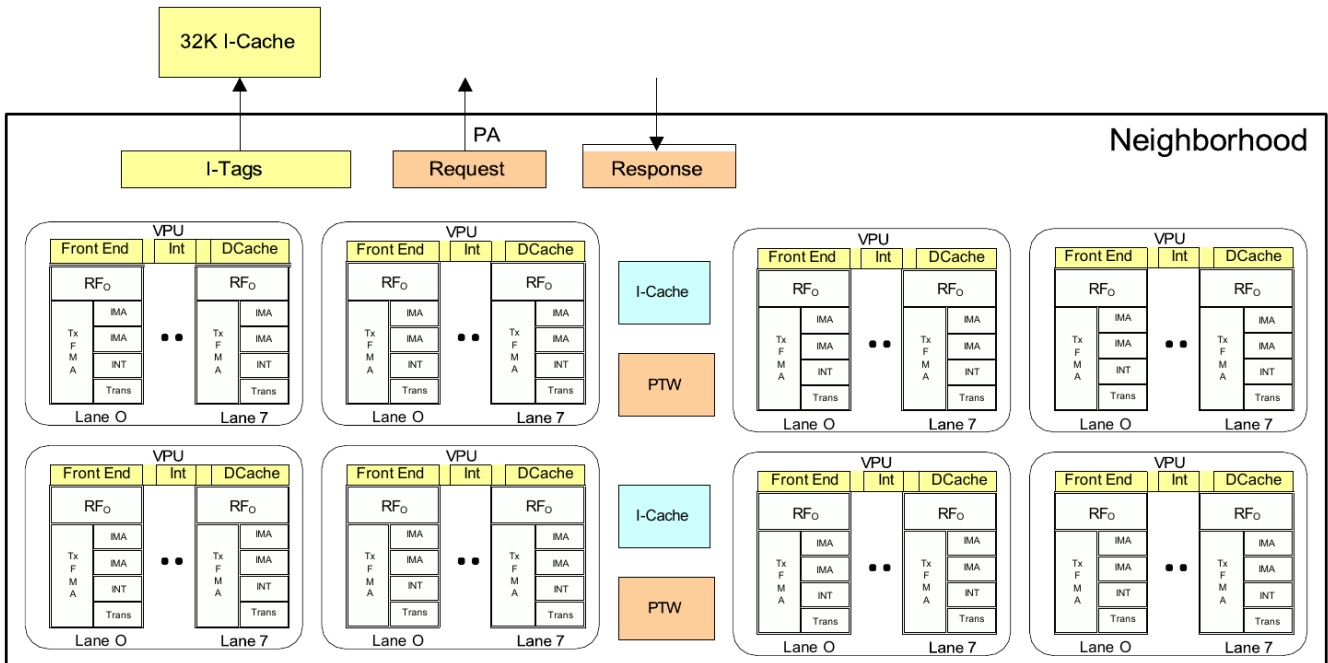


Figure 3. Neighborhood

1.2.3. Minion Shire

A Minion Shire groups four Neighborhoods (Figure 4). It also contains:

- **Shire Cache:** 4 MB, in four 1 MB banks. Programmable ESRs divide the cache into three kinds of partition:
 - L2 cache, private to the Shire
 - Slice of the chip-wide distributed L3 cache
 - Scratchpad memory
- **Crossbar:** Request and response crossbars connect four Neighborhoods and the RBOX port to the four Shire Cache banks and the UC block. Each Neighborhood has one request and response port into the Shire Cache, and each bank accepts at most one request per cycle. The crossbar also connects the Neighborhoods to the un-cached (UC) block.
- **Uncached (UC) block:** receives the Neighborhood requests that are not sent to the Shire Cache banks. It handles uncacheable accesses, contains the fast local barrier (FLB) and fast local credit counter (FCC) units, and delivers messages sent between ET-Minion cores of different Shires.
- **ESR banks:** the Esperanto System Registers (ESRs) configure the blocks in the Shire.
- **NoC interface:** L2 misses and uncacheable accesses leave the Shire here, bound for other Shires or the memory controllers.

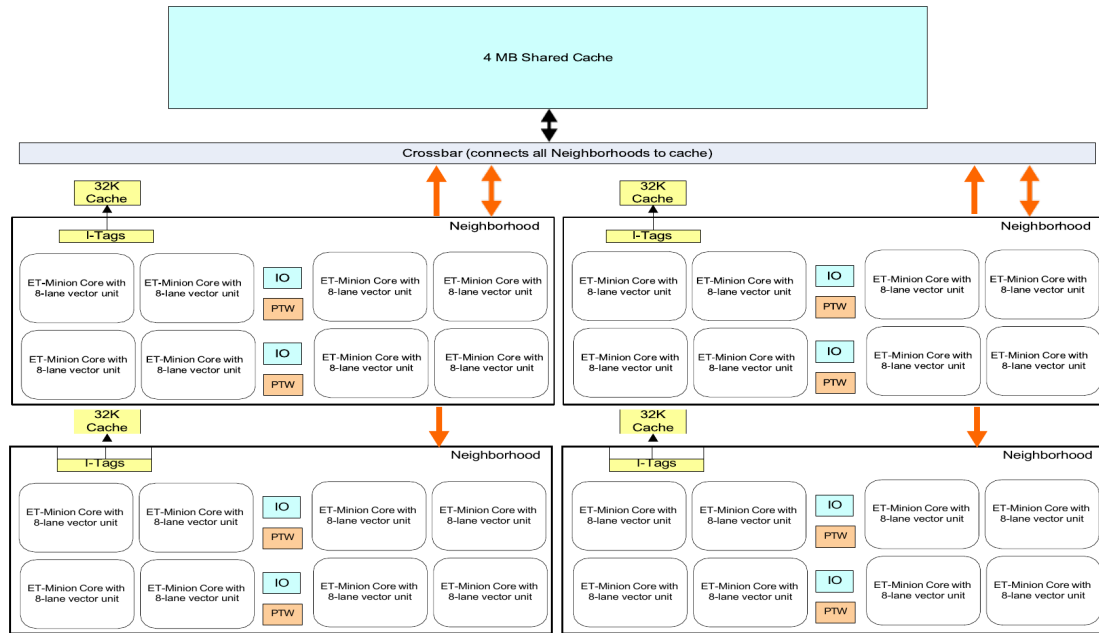


Figure 4. Minion Shire

1.2.4. Minion Shire Array

The 34 Minion Shires, 1,088 ET-Minion cores in total, form the Minion Shire array. The roles of the Shires are assigned by software:

- **Compute:** 32 Minion Shires operate in parallel as the compute array.
- **Master (management):** one Minion Shire receives work and distributes it to the compute array.
- **Spare:** one Minion Shire.

All Minion Shires are identical in hardware, so any Minion Shire can take any of these roles.

1.3. ET-Maxion and Service Processor

In addition to the ET-Minion cores, the device contains two other core types. Both are located in the I/O Shire ([Section 1.4.4](#)). This document does not describe them beyond this section.

1.3.1. ET-Maxion Cores

The ET-Maxion is a single-threaded, superscalar, out-of-order 64-bit RISC-V core intended for management and general-purpose tasks. The device contains four identical ET-Maxion cores. Together with a 4 MB shared L2/L3 cache they form the Maxion Neighborhood, and a system bus and coherency hub in the I/O Shire connects them. [Figure 5](#) shows the ET-Maxion pipeline.

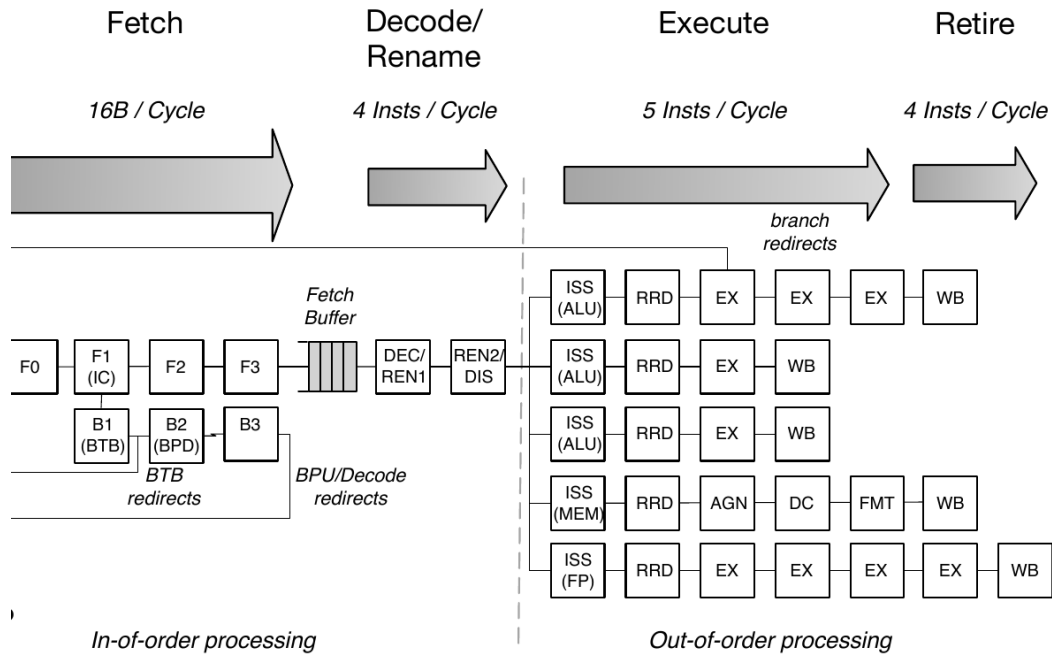


Figure 5. ET-Maxion pipeline

1.3.2. Service Processor

The Service Processor (SP) is a single-threaded ET-Minion core without the vector/tensor extensions. It has its own ROM and a 1 MB scratchpad SRAM, but none of the Neighborhood or Shire shared resources.

- **Boot.** The SP is the first major block released from reset. It executes the boot code from ROM and initializes the device interfaces, including DDR, PCIe and USB.
- **Run time.** It provides management services for the whole device.
- **Access.** The SP can access every resource on the device, and some resources are accessible only to it.

The I/O Shire also contains a hardware root of trust, and six secure mailboxes for communication between four agents:

- Service Processor
- Master Minion Shire

- ET-Maxions
- PCIe host

1.4. Mesh Network-on-Chip

The mesh NoC is the top-level interconnect of ET-SoC-1 and connects all Shires. Figure 6 shows the device with the Shire array on the mesh, the external memory and PCIe interfaces, and the peripheral, Service Processor and debug networks.

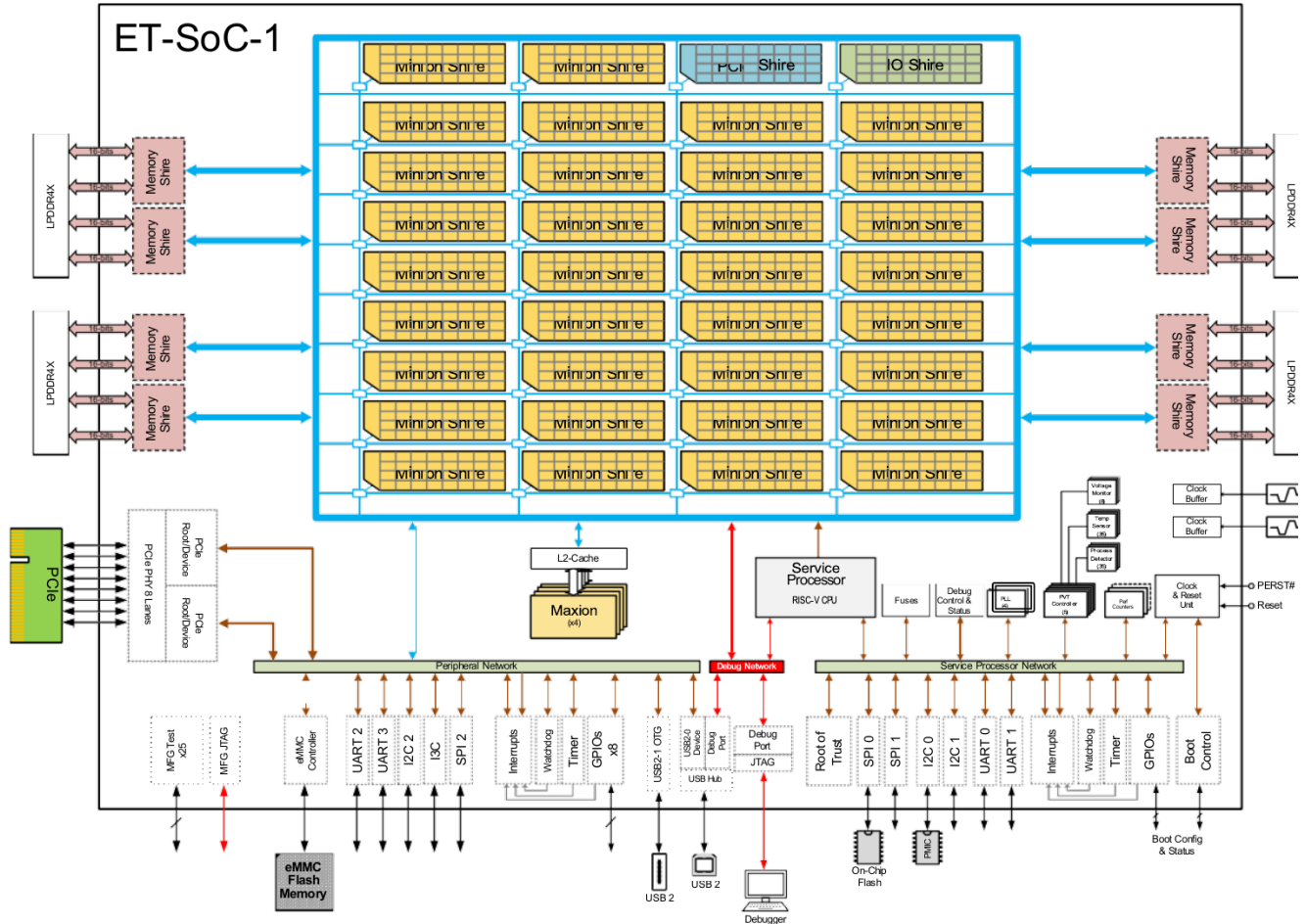


Figure 6. ET-SoC-1 block diagram

1.4.1. Mesh Topology and Mesh Stops

The mesh is organized as an 8×6 grid, and each Shire attaches to it at a mesh stop. The Memory Shires occupy the east and west edges, four on each side, which leaves the corners of the grid unoccupied. The result is 44 mesh stops, listed in Table 2.

Table 2. Shires on the mesh

Shire	Count	Contents
Minion Shire	34	32 ET-Minion cores and a 4 MB Shire Cache
Memory Shire	8	LPDDR4X memory controller
PCIe Shire	1	PCIe controllers and PHY

Shire	Count	Contents
I/O Shire	1	ET-Maxions, Service Processor and peripherals
Total	44	

1.4.2. Memory Shires

Each Memory Shire contains a 32-bit LPDDR4X PHY and two memory controllers, each driving one 16-bit channel achieving 4,266 MT/s/channel. The Memory Shires also support global atomic operations.

1.4.3. PCIe Shire

The PCIe Shire contains two dual-mode PCIe Gen4 controllers that share an 8-lane PHY. It supports three lane configurations:

- Single 8-lane endpoint or root complex
- Single 4-lane endpoint or root complex
- 4-lane endpoint together with a 4-lane root complex

The interface is normally factory-configured as an endpoint.

1.4.4. I/O Shire

The I/O Shire contains the ET-Maxions and the Service Processor ([Section 1.3](#)). It also contains the device peripherals, which are grouped on three networks according to which agents may access them.

- **Service Processor network.** Accessible only to the SP. It contains:
 - SPI, including the boot-flash controller
 - I2C, including the link to the external power-management controller
 - UART, including the SP console
 - GPIO, timers, watchdog and boot control
- **Peripheral network.** Shared by the SP, the ET-Maxions and the master Minion Shire. It contains:
 - eMMC, USB 2.0 OTG, I2C/SMBus, I3C, SPI and UART
 - GPIO, timers and watchdog
- **Debug network.** Provides debug access through JTAG and a USB 2.0 device port.

1.5. ET-SoC-1 Accelerator Card

ET-SoC-1 operates as an accelerator attached to a host system over PCI Express. It is mounted on a PCIe add-in card ([Figure 7](#)) with the following features:

- PCIe x16 card-edge connector, with an 8-lane Gen4 interface to ET-SoC-1

- ET-SoC-1 soldered to the card
- Four LPDDR4X devices forming a 256-bit memory interface
- 64 GB eMMC flash memory
- USB connectors for the ET-SoC-1 device and debug ports and for a UART interface, and a JTAG connector
- DIP switches that select the boot options

Multiple ET-SoC-1 devices can also be integrated on a single card.



Figure 7. ET-SoC-1 PCIe card

2. ET-Minion Core

2.1. Core Overview

This section describes the internal organization of the ET-Minion core: its block partitioning, hardware threads and instruction flow. The features of the core and its position in the ET-Minion hierarchy are summarized in [Section 1.2.1](#).

The ET-Minion is a dual-threaded, in-order, single-issue 64-bit RISC-V core that implements RV64IMFC with the Esperanto instruction extensions. It is implemented in `minion_top` and consists of four units: the Front End, the integer pipeline, the L1 data cache and the VPU. The core fetches instructions from the Neighborhood L0 micro-cache and accesses memory through the Neighborhood memory path; it has no instruction cache of its own.

2.1.1. Block Partitioning

The ET-Minion core is implemented in `minion_top` ([Figure 2](#)). `minion_top` contains two blocks: `core_top`, which holds the Front End, the integer pipeline, the L1 data cache and the debug APB slave, and `vpu_top`, the vector processing unit. [Table 3](#) lists the blocks.

Table 3. ET-Minion core blocks

Block	Module	Function
Front End	<code>frontend_top</code>	Fetches instructions for both threads from the Neighborhood L0 micro-cache, expands compressed instructions, decodes them for the integer pipeline and the VPU, and issues one instruction per cycle to the integer pipeline (Section 2.2).
Integer pipeline	<code>intpipe_top</code>	In-order, single-issue pipeline with the stages ID, EX, GSC, TAG, MEM and WB. Contains the integer register file, ALU, multiply/divide unit, integer, FP and mask scoreboards, and the CSRs. Redirects the Front End on a change of control flow.
Data cache	<code>dcache_top</code>	4 KB private L1 data cache, partly configurable as scratchpad memory. Contains the data TLB, PMA check, miss handling, replay queue, store merge, atomic (AMO) forwarding to L2 through the miss handler's UC flow, scratchpad controller, write-back unit, cache management, tensor-load and reduce units, and the interface to the Neighborhood memory path.
APB slave for debug registers	<code>minion_debug_apb_slv</code>	APB slave for debug access to the core, including writes to the debug program buffer.
VPU	<code>vpu_top</code>	Vector processing unit with a control block and eight identical lanes. Executes floating-point, packed-integer, transcendental and tensor operations.

2.1.2. Hardware Threads

The core runs two hardware threads (`CORE_NR_THREADS` = 2). Each thread has its own architectural state; the pipeline and execution units are shared, and instructions of the two threads are interleaved in the pipeline.

Table 4. Per-thread and shared resources

Per thread	Shared by both threads
Fetch PC, thread buffer and instruction FIFO in the Front End	Front End request port, decoders and thread scheduler
32 x 64-bit integer registers, in one register file indexed by thread	Integer pipeline, ALU and multiply/divide unit
32 vector registers, (also the FP f-registers), 256 bits wide, held as a 32-bit slice in each lane's RF, which are indexed by the thread number. Eight 8-bit mask registers.	VPU control and execution units
CSR state, interrupt inputs and debug state	L1 data cache and scratchpad

Each thread is enabled by its corresponding bit of the `enabled` input of `minion_top`. `minion_top` registers `enabled` once, and the integer pipeline and debug logic uses this registered copy directly. The Front End registers it twice more, so that it can detect the edges of the enable and turn each rising and falling edge into a one-cycle enable of disable event ([Section 2.2.6.3](#)). When a thread is enabled, it starts fetching at `reset_vector`, which is the same for both threads.

The Front End selects one thread per cycle for issue to the integer pipeline. The integer pipeline can stall each thread independently, for example on `WFI`, a fence, a scoreboard hazard, or while the other thread holds exclusive mode. As a result, the Front End sees a change of `enabled` two cycles later than the integer pipeline does.

2.1.3. Instruction Flow Through the Core

The Front End fetches instructions for both threads from the Neighborhood L0 micro-cache, expands compressed instructions, decodes them, and presents one instruction per cycle to ID, the first stage of the integer pipeline. The integer pipeline executes the instructions in order and passes memory operations to the data cache and vector operations to the VPU. In the opposite direction, the integer pipeline redirects the Front End to a new fetch address after a branch misprediction, exception or other change of control flow, and stalls a thread that must not issue.

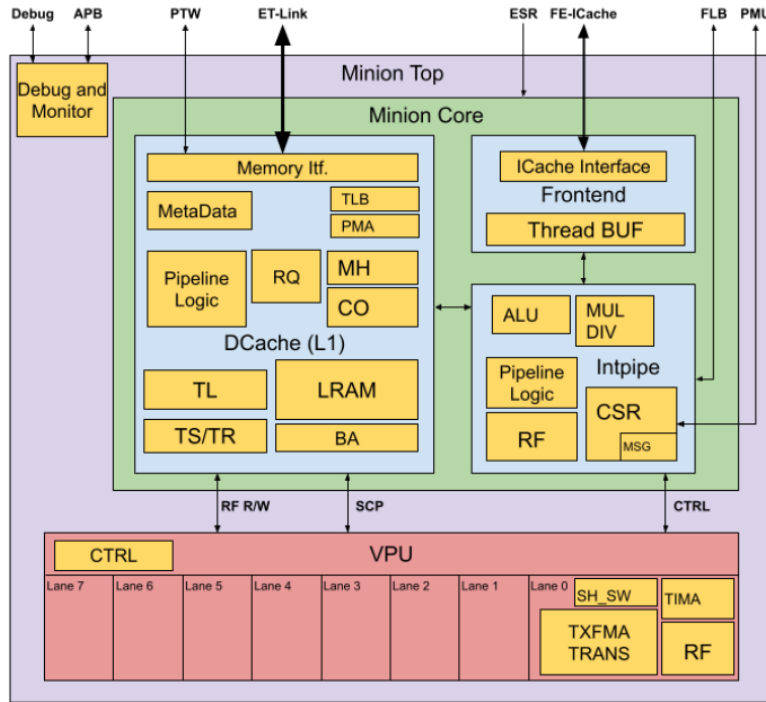


Figure 8. Front End in the ET-Minion core

2.1.4. Interfaces to the Neighborhood

Table: ET-Minion core interface groups.

2.1.5. Clocks, Resets and Power

2.1.6. Debug Support

2.2. Front End

2.2.1. Overview

The Front End (`frontend_top`) supplies instructions to the ET-Minion core. It sits between the Neighborhood L0 micro-cache and the ID stage of the integer pipeline. It holds the program counter of each of the two threads and fetches 32-byte instruction blocks from the L0 micro-cache. It then extracts the instruction, expands compressed instructions to 32 bits and decodes them for both the integer pipeline and the VPU. It delivers one decoded instruction per cycle to ID, from either thread. The core has no instruction cache of its own. Each L0 micro-cache is shared by four ET-Minion cores, eight threads in all, so the design of the Front End is driven by the need to hide the latency of shared fetch path.

The Front End is built around the following four concepts:

1. **Fetch is double-buffered:** each thread has a thread buffer with two 256-bit entries, each holding half a cache line, so the next block can be requested while the current one is being consumed.
2. **Each thread has at most one fetch request in flight:** the L0 micro-cache answers every accepted request, hit or miss, after a fixed latency, so a thread recognizes its own by timing

alone, without a thread identifier. On a miss the thread sleeps until the L0 micro-cache reports that a fill has completed, and then retries.

3. **Instructions are decoded before they are queued:** one integer decoder and one VPU decoder, shared by both threads, decode in F7, and each thread has a two-entry instruction FIFO that holds decoded instructions until ID accepts them. This moves decode out of the ID stage and lets a thread keep fetching while ID serves the other thread.
4. **The two threads are independent:** each thread fetches, sleeps, stalls and restarts on its own. They share only the fetch request port, the decoders and the issue slot into ID, and the thread scheduler alternates between them when both are eligible.

The Front End is a pipeline of eight stages, F0 to F7. F1 to F5 overlap the L0 micro-cache pipeline. With an L0 hit and no competing requests, the first instruction of a block reaches ID nine cycles after it is requested. The first target instruction reaches ID ten cycles after a redirect, and the first instruction reaches ID eleven cycles after a thread is enabled.

The integer pipeline controls the Front End through two per-thread signals. A redirect (`f0_core_req_valid`, `f0_core_req`) flushes the thread and restarts fetch at a new PC. It is used for mispredictions, exceptions, replays, interrupts and debug-mode transitions. A stall (`id_core_stall`) stops the thread from being scheduled, for example on WFI, a fence, a CSR access that must wait, or a scoreboard hazard. The Front End attaches the fetch status to each instruction: page and access faults, bus and ECC errors, and a replay flag for uncacheable fetches issued while older instructions are still in flight. The integer pipeline acts on this status in ID.

The Front End also starts and stops threads through their enable inputs and reset vector. It supports debug in two ways: it generates the placeholder instruction on which a halt is taken, and it lets entry 0 of a halted thread's buffer serve as the debug program buffer. Each thread buffer runs on its own gated clock; a chicken bit disables the gating.

Table 5 lists the submodules and Figure 9 shows how they are connected. The sections that follow cover the interface (Section 2.2.2), the pipeline stages (Section 2.2.3), the fetch protocol with the L0 micro-cache (Section 2.2.4), the thread buffer (Section 2.2.5 and Section 2.2.6), decode (Section 2.2.7), the instruction FIFO (Section 2.2.8), the thread scheduler (Section 2.2.9), and halt and debug support (Section 2.2.10).

Table 5. Front End submodules

Submodule	Instances	Function
Thread buffer (<code>frontend_thread_buffer</code>)	2, one per thread	64-byte instruction buffer and the thread's program counter. Issues fetch requests to the L0 micro-cache.
RVC expander (<code>frontend_rvc_expander</code>)	2, one per thread buffer	Expands compressed (RVC) instructions to 32 bits.
Request arbiter (<code>arb_lru_data</code>)	1	2:1 LRU arbiter. Selects the thread whose fetch request is sent to the L0 micro-cache.
Integer decoder (<code>intpipe_decode</code>)	1	Decodes the instruction into integer-pipeline control signals.

Submodule	Instances	Function
VPU decoder (<code>vpu_decoder</code>)	1	Decodes the instruction into VPU control signals.
Instruction FIFO (inline generate block)	2, one per thread	Two-entry queue of decoded instructions waiting for the integer pipeline.
Thread scheduler (<code>frontend_thread_sched</code>)	1	Selects the thread whose instruction is presented to the integer pipeline.

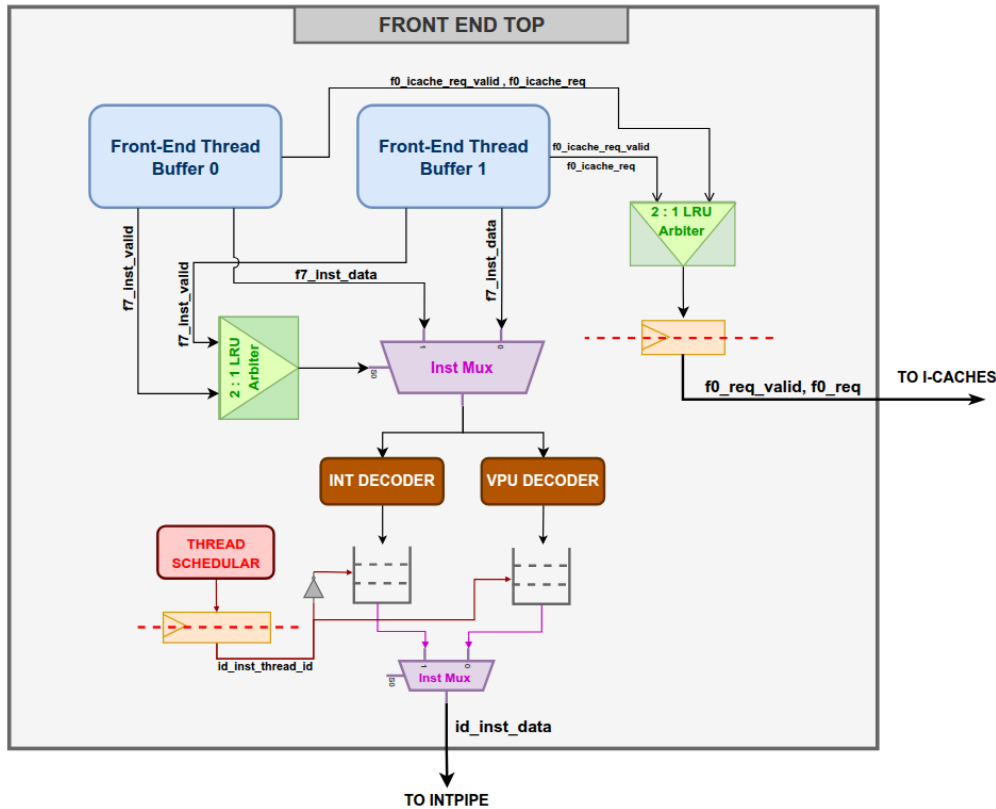


Figure 9. Front End block diagram

2.2.2. Interface

This section defines the boundary of `frontend_top`. Section 2.2.2.1 lists its ports in six groups. The system group has the clock, the warm and debug resets and the clock-gating chicken bit. Thread control has the thread enables, the reset vector and the virtual-memory status of each thread. The L0 micro-cache group has the fetch request and response and the fill-done signal. The integer-pipeline group has the per-thread redirect and stall inputs. The output group carries the instruction and it's decoded data to ID. The debug group has the halt, halted and program-buffer signals. For each port the table gives the net it connects to in `core_top`.

Section 2.2.2.2 gives the field layout and bit width of each packed structure that crosses these ports. These are the fetch request and response, the virtual-memory status, the redirect, the instruction delivered to ID, the instruction FIFO entry, and the two decoded control words. Section 2.2.2.3 lists the defines that set the number of threads, the buffer geometry and the PC width.

2.2.2.1. I/O Signals

Table 6 lists the ports of `frontend_top`. *Connected to* gives the signal in `core_top`. A width of 2 denotes one bit per thread.

Table 6. Front End I/O signals

Signal	Dir	Width / type	Connected to	Description
System				
<code>clock</code>	in	1	<code>clock</code>	Core clock. Each thread buffer derives a gated clock from it.
<code>reset</code>	in	1	<code>reset_w</code>	Warm reset. Synchronous.
<code>reset_debug</code>	in	1	<code>reset_d</code>	Debug reset. Clears the program-buffer control and, while the thread is halted, refills the program buffer with <code>EBREAK</code> .
<code>chicken_bit</code>	in	1	<code>chicken_bit_frontend</code>	Disables Front End clock gating. Driven by the <code>min_frontend_clock_gate_disable</code> chicken bit.
Thread control				
<code>f0_thread_enabled</code>	in	2	<code>enabled</code>	Thread enable. A rising edge starts fetch at the reset vector; a falling edge stops fetch.
<code>f0_reset_vector</code>	in	48	<code>reset_vector</code>	Fetch address after reset or thread enable.
<code>vm_status</code>	in	2 x <code>minion_vm_status</code>	<code>vm_status</code>	Virtual-memory status of each thread, forwarded with every fetch request.
Fetch request to the L0 micro-cache				
<code>f1_icache_req_valid</code>	out	1	<code>icache_req_valid</code>	The F1 I-cache request register holds a request.
<code>f1_icache_req</code>	out	<code>frontend_icache_req</code>	<code>icache_req</code>	Fetch request: thread, VM status and address.
<code>f1_icache_req_ready</code>	in	1	<code>icache_req_ready</code>	The Neighborhood accepts the request in this cycle.
Fetch response from the L0 micro-cache				
<code>f5_icache_resp_valid</code>	in	1	<code>icache_resp_valid</code>	A response is present. It carries no thread identifier.

Signal	Dir	Width / type	Connected to	Description
f5_icache_resp_miss	in	1	icache_resp_miss	The response is a miss and carries no instruction data.
f5_icache_resp	in	icache_frontend_resp	icache_resp	32-byte block and status.
f6_icache_fill_done	in	1	icache_fill_done	The L0 micro-cache has completed a line or translation fill.
Redirect and stall from the integer pipeline				
f0_core_req_valid	in	2	id_core_fe_req_valid	Redirect. Flushes the thread and restarts fetch at f0_core_req.pc.
f0_core_req	in	2 x minion_fe_req	id_core_fe_req	Redirect PC, speculation status and debug-mode transition.
id_core_stall	in	2	id_core_fe_stall	Thread must not issue. Raised for WFI, CSR accesses that are replayed until a resource is ready (e.g., TensorWait, tensor ops, FCC), a fence, exclusive mode held by the other thread, and integer or FP scoreboard stalls. The WFI/CSR term is suppressed while debug mode or halt request is active.
Instruction to the integer pipeline (ID)				
id_inst_valid	out	1	id_fe_core_resp_valid	The instruction presented to the integer pipeline is valid.
id_inst_ready	in	1	id_core_fe_resp_ready	The integer pipeline is ready to accept the instruction.
id_inst_thread_id	out	1	id_fe_core_resp_thread_id	Thread of the instruction.
id_inst_data	out	frontend_core_resp	id_fe_core_resp	Instruction, PC and fetch status.
id_intpipe_ctrl	out	minion_control	id_intpipe_ctrl	Integer decode.
id_vpu_decoder_sigs	out	vpu_ctrl_sigs_t	id_vpu_decoder_sigs	VPU decode. Valid only for VPU instructions.

Signal	Dir	Width / type	Connected to	Description
id_vpu_core_ctrl	out	vpu_minion_id_ctrl	id_frontend_vpu_ctrl	VPU and mask register usage, for hazard checks in the integer pipeline.
Debug				
halt	in	2	debug_in_core.halt	A halt request is pending.
halted	in	2	debug_out.halted	The thread is in debug mode.
debug_ffb_wdata	in	64	debug_ffb_wdata	Program-buffer write data.
debug_ffb_en	in	4	debug_ffb_en	Program-buffer word write enables.
debug_ffb_thread_sel	in	1	debug_ffb_thread_sel	Thread whose program buffer is written.
debug_ffb_exec	in	2	debug_ex_program_buffer	Start executing the program buffer.

2.2.2.2. Data Structures

Table 7. `frontend_icache_req`: fetch request (58 bits)

Field	Width	Description
thread_id	1	Requesting thread.
vm_status	minion_vm_status	Virtual-memory status of the thread (Table 8).
addr	49	Fetch address. The L0 micro-cache returns the aligned 32-byte block that contains it.

Front End addresses are 49 bits wide: the 48-bit virtual address and one additional bit that allows a non-canonical 64-bit address to be represented.

Table 8. `minion_vm_status`: virtual-memory status (8 bits)

Field	Width	Description
prv	2	Current privilege level.
mprv, mpp, sum, mxr	1, 2, 1, 1	Corresponding <code>mstatus</code> fields.
debug	1	Thread is in debug mode.

Table 9. `icache_frontend_resp`: fetch response (261 bits)

Field	Width	Description
data	256	32-byte instruction block.
page_fault	1	Instruction page fault.
access_fault	1	Instruction access fault.

Field	Width	Description
cacheable	1	Block is in a cacheable region.
bus_err	1	Bus error on the line.
ecc_err	1	ECC error on the line.

Table 10. `minion_fe_req`: redirect (52 bits)

Field	Width	Description
pc	49	New fetch PC.
speculative	1	Thread has older instructions in the integer pipeline. Driven every cycle, independently of <code>f0_core_req_valid</code> . Used for replay (Section 2.2.5.6).
debug_info.halt	1	Redirect enters debug mode.
debug_info.resume	1	Redirect leaves debug mode.

Table 11. `frontend_core_resp`: instruction to the integer pipeline (89 bits)

Field	Width	Description
pc	49	PC of the instruction.
inst_bits	32	Instruction. Compressed instructions are delivered expanded.
pf0, af0	1 each	Page fault / access fault on the entry holding the start of the instruction.
pf1, af1	1 each	Page fault / access fault on the entry holding the upper half of an instruction that crosses entries.
replay	1	Instruction must be replayed.
rvc	1	Instruction was compressed.
bus_error, ecc_error	1 each	Bus error / ECC error on the entry or entries holding the instruction.

Table 12. `frontend_thread_data`: instruction FIFO entry (179 bits)

Field	Width	Description
core_resp	89	Instruction and fetch status (Table 11).
vpu_ctrl_sigs	45	VPU decode (<code>vpu_ctrl_sigs_t</code>).
intpipe_ctrl	45	Integer decode (<code>minion_control</code>).

Table 13. Decoded control words

Structure	Width	Content
<code>minion_control</code>	45	Legal instruction; software-emulated (M-code) instruction; branch, <code>JAL</code> , <code>JALR</code> ; integer source reads and destination write; ALU operand selects, function and width; immediate format; memory operation, command and size; FP/VPU destination write; mask register read/write; multiply/divide; CSR command; fence; global memory access; gather/scatter; graphics; implicit use of <code>x31</code> .
<code>vpu_ctrl_sigs_t</code>	45	VPU function; execution unit (TxFMA, shift/swizzle, transcendental ROM); conversion and load/store type; source, third-source and destination reads and write; mask register reads; operand swaps; scalar or packed; data type; integer-register transfer; rounding; FP flag update.

2.2.2.3. Configuration Parameters

Table 14. Front End configuration parameters

Define	Value	Description
<code>CORE_NR_THREADS</code>	2	Threads per core.
<code>FE_FETCH_BUFFERS</code>	2	Thread-buffer entries per thread.
<code>FE_FETCH_READ_SIZE</code>	256	Bits per fetch block and per entry.
<code>FE_FETCH_PTR_SIZE</code>	5	Read-pointer width: 1 entry bit and 4 halfword bits. $\log_2(\text{FE_FETCH_VALID_INST_SIZE} * \text{FE_FETCH_BUFFERS}) = \log_2(16 * 2) = 5$
<code>PC_SIZE_EXT</code>	49	Front End PC width.

2.2.3. Pipeline Stages

This section describes the eight Front End stages, F0 to F7, and how they line up with the L0 micro-cache pipeline. Two activities run in parallel and meet in the thread buffer. Fetch runs from F0 to F6: a thread's request is arbitrated and sent in F0, waits for acceptance in F1, is looked up in F1 to F5, and the response is written into the thread buffer in F6; a miss writes nothing and the thread goes to sleep. In F7 each thread pulls its next instruction out of its buffer and expands it if compressed. One thread's instruction is decoded and placed in its FIFO. The scheduler presents one thread's FIFO head to the integer pipeline each cycle

Fetching and extracting are independent: a thread keeps extracting instructions from a filled buffer entry while its next request is still in F0-F5.

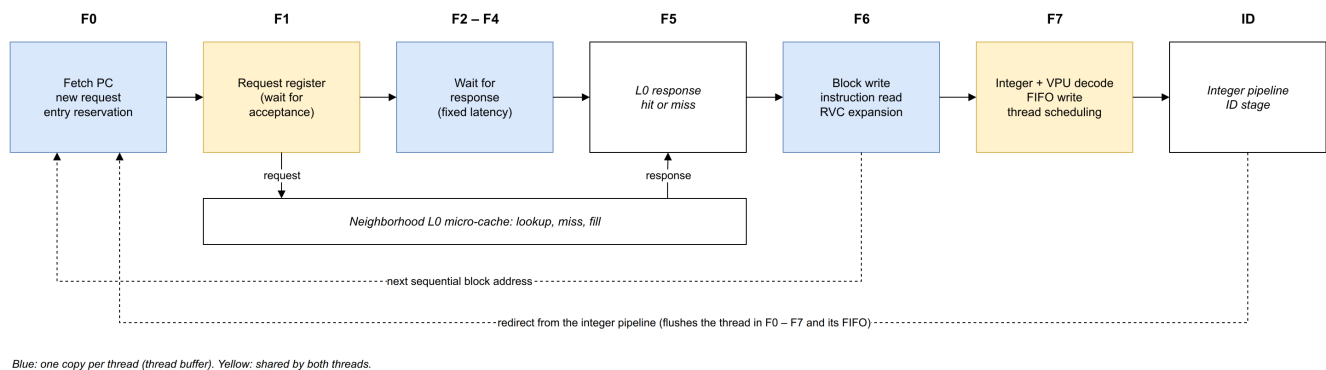


Figure 10. Front End pipeline

F0 Stage

The F0 stage receives instruction-block requests from both thread buffers and decides which one gets access to the L0 micro-cache using a 2:1 LRU arbiter. The selected request is registered and sent to the L0 micro-cache through the Neighborhood interface. This stage also handles the enabling and disabling of threads and redirect requests from the integer pipeline, for example the new address after a mispredicted branch.

F1–F5 Stage

The selected request is held in F1 until the L0 micro-cache accepts it, which happens at the earliest in the second F1 cycle. The thread then waits for the response. The L0 micro-cache always responds to an accepted request, whether it is a hit or a miss, with a fixed five-cycle latency: counting the cycle in which it accepts the request as the first cycle, the response is returned in the fifth cycle, in F5 (Figure 12).

F6 Stage

The response from the L0 micro-cache is checked. If it is a hit, the instruction block is stored in the buffer of the thread that issued the request. If it is a miss, nothing is stored, and the thread waits for the cache to be filled. Instruction reading is independent of the fetch request: whenever a buffer entry holds instructions, the thread reads its next instruction, expands it if it is compressed, and passes it on for decoding, in parallel with any request in F0 to F5.

F7 Stage

This stage selects which thread's instruction is read from its buffer. The selected instruction is decoded and stored, along with its control signals, in that thread's FIFO. The thread scheduler decides which thread's FIFO issues an instruction to the integer pipeline in the next cycle. If both threads are eligible, they are selected in round-robin order; otherwise, the eligible thread is selected.

With an L0 hit and no competing requests, the first instruction of a block reaches ID nine cycles after its request is issued in F0.

2.2.4. Interaction with the L0 Micro-Cache

This section describes the fetch protocol between the Front End and the L0 micro-cache, from the Front End side. It covers the following four things:

1. How a request is arbitrated and accepted: by the 2:1 LRU arbiter and F1 request register in the

core, then by the 8:1 arbiter of the Neighborhood.

2. The fixed response timing, and how a thread recognizes its own responses with a token tracker, since responses carry no thread identifier.
3. The miss, sleep and fill-done protocol, and why its timing guarantees that no wake-up is lost.
4. How a redirect cancels requests already in flight, including one still waiting in the F1 register.

Each L0 micro-cache serves the eight threads of four ET-Minion cores. [Figure 11](#) shows the request and response path of one core. The Neighborhood side of the interface is described in [Section 3.2](#).

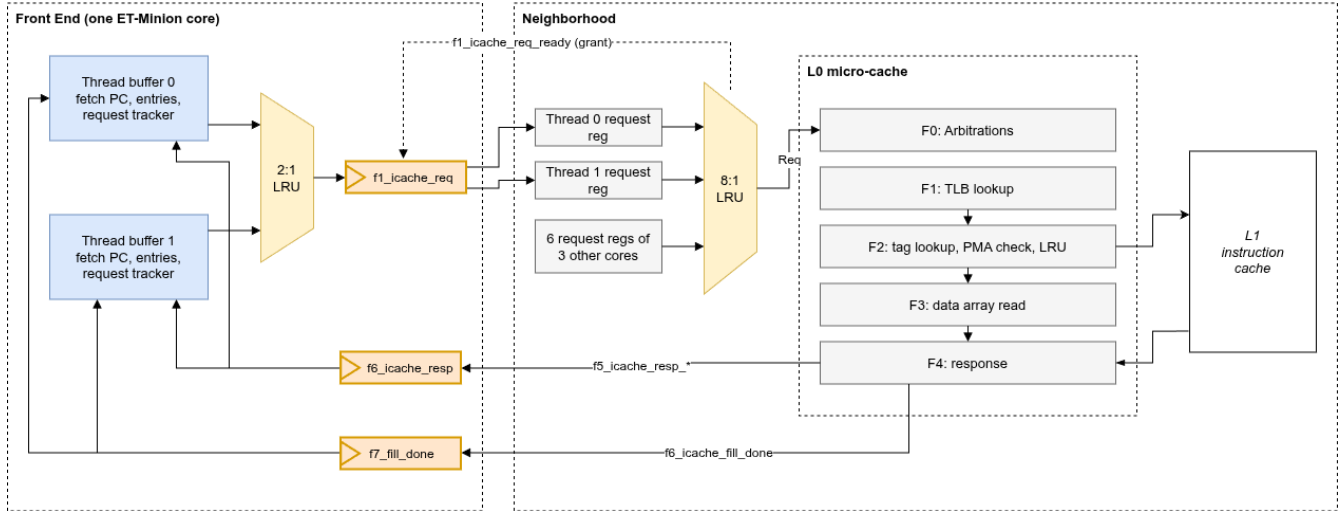


Figure 11. Front End to L0 micro-cache request and response path

2.2.4.1. Request Arbitration and Acceptance

A fetch request passes through the following steps:

1. **2:1 LRU arbiter, F0.** Selects between the two threads of the core. With two clients the grant alternates when both threads request. The priority is updated on every grant.
2. **F1 I-cache request register (f1_icache_req).** Holds the granted request until the Neighborhood accepts it with `f1_icache_req_ready`. The register accepts a new request when it is empty or its current request is accepted in the same cycle; otherwise the 2:1 arbiter issues no grant.
3. **8:1 LRU arbiter, Neighborhood.** The request is first captured in a per-thread request register, then arbitrated against the other seven threads served by the same L0 micro-cache. The grant of this arbiter is returned as `f1_icache_req_ready`. Because of the capture register, a request is accepted at the earliest in its second F1 cycle.

2.2.4.2. Response Timing and Ownership

The L0 micro-cache has a five-stage pipeline and returns exactly one response per accepted request, from its last stage, for a hit and for a miss alike. The response reaches the Front End in F5.

The response carries no thread identifier. Each thread buffer identifies its responses by position in time: when its request is granted, the thread inserts a token (a valid bit and the reserved entry number) into its own F1–F6 tracker. The token remains in F1 until the request is accepted and then advances one stage per cycle. A response registered in F6 together with a valid token belongs to that thread, and the token selects the entry to be written. The response and fill-done signals are

registered in `frontend_top` and broadcast to both thread buffers.

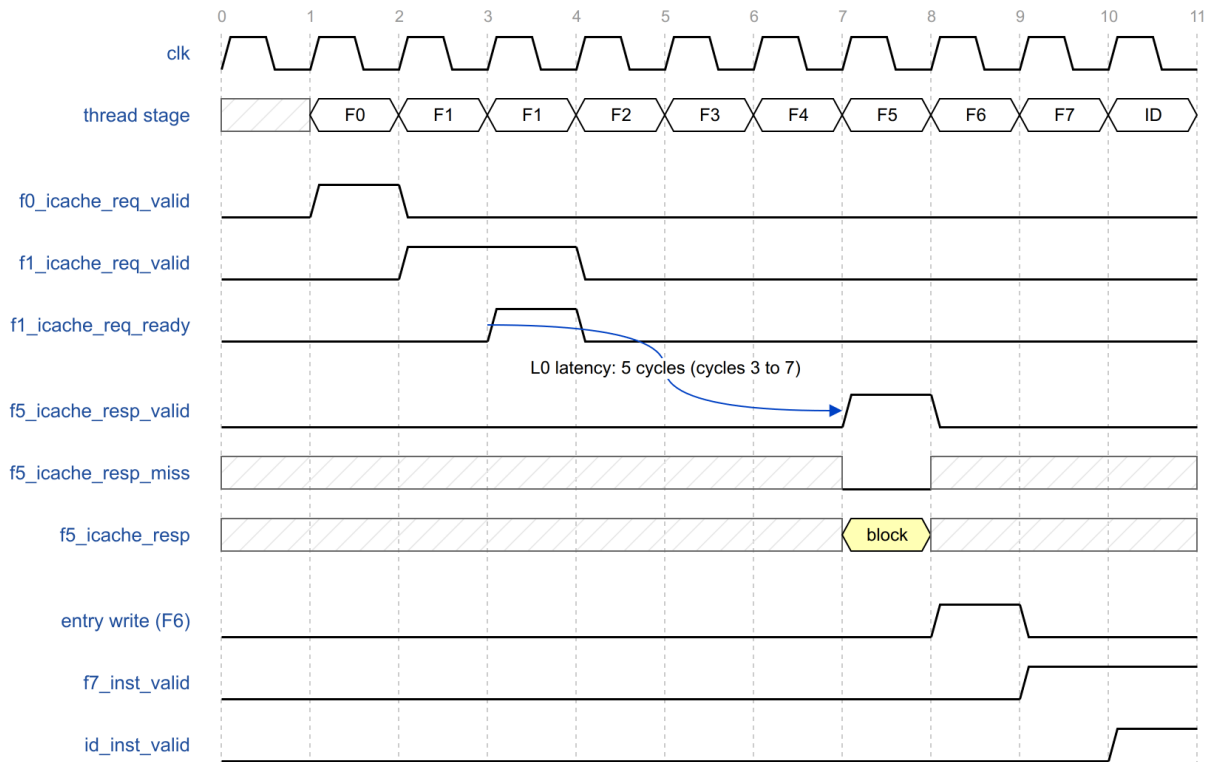


Figure 12. Request and hit response

In Figure 12 the request is issued in cycle 1, accepted in cycle 3, and the block returns in cycle 7. It is written in cycle 8; the first instruction of the fetched block is in the F7 stage in cycle 9 and in the ID stage in cycle 10.

2.2.4.3. Miss Handling and Fill Done

A miss response has the same timing as a hit and carries no data. On a miss the thread writes nothing, keeps its fetch PC and enters a sleep state (`f0_miss_pending`) in which it issues no requests.

On completion of a line fill from the L1 instruction cache, or of a translation fill, the L0 micro-cache pulses `f6_icache_fill_done`. The signal is broadcast to all threads of the L0 micro-cache and carries no address. Every sleeping thread wakes and retries; a thread whose line is still absent misses again and sleeps until the next fill. A redirect, a thread disable or a pending halt also clears the sleep state.

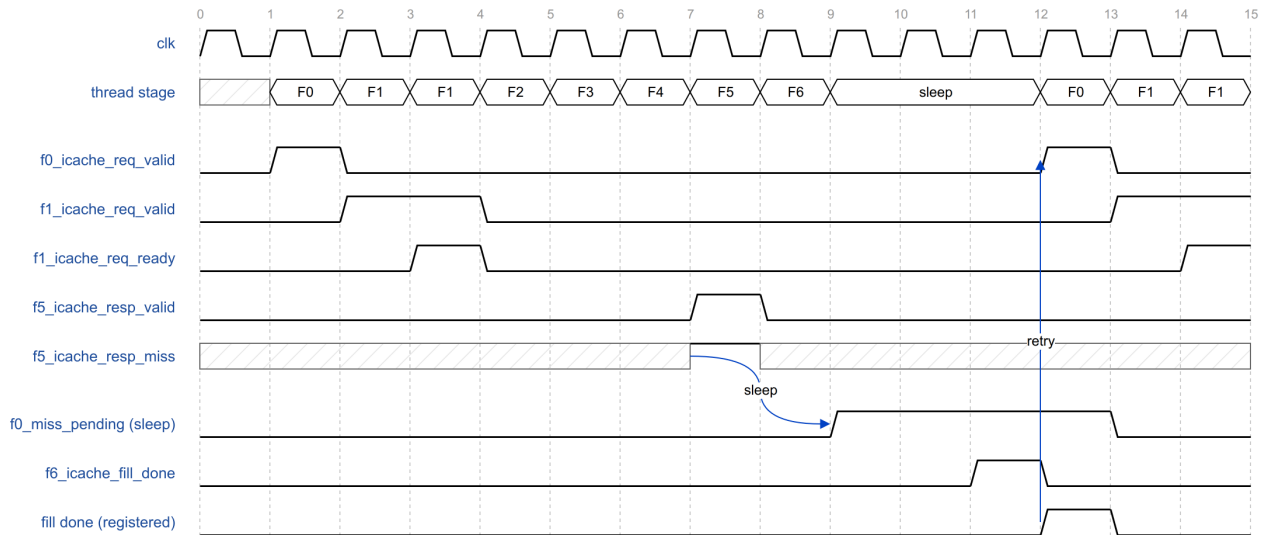


Figure 13. Miss, sleep and retry

In Figure 13 the miss response arrives in cycle 7 and the thread sleeps from cycle 9. Fill done arrives in cycle 11, is registered, and the thread retries in cycle 12.

Fill done is issued one cycle after the response slot. A request that looks up the L0 micro-cache while a fill is being written still misses, and its response is the last miss that the fill can cause. Issuing fill done one cycle after that response, and registering it again in the Front End, ensures that every thread has entered the sleep state before the wake-up arrives. A wake-up that arrived first would be lost, and the thread would wait for an unrelated fill.

2.2.4.4. Request Cancellation

A redirect invalidates the thread's tokens in F2 to F5 and suppresses the F6 write, so the responses of in-flight requests are discarded.

A request still waiting in the F1 I-cache request register cannot be withdrawn: the register is shared and the Neighborhood already holds a copy. The thread records the cancellation and does not advance the token when the request is accepted. The I-cache lookup is performed and its response is discarded. The thread's new request is issued only after the cancelled request has been accepted.

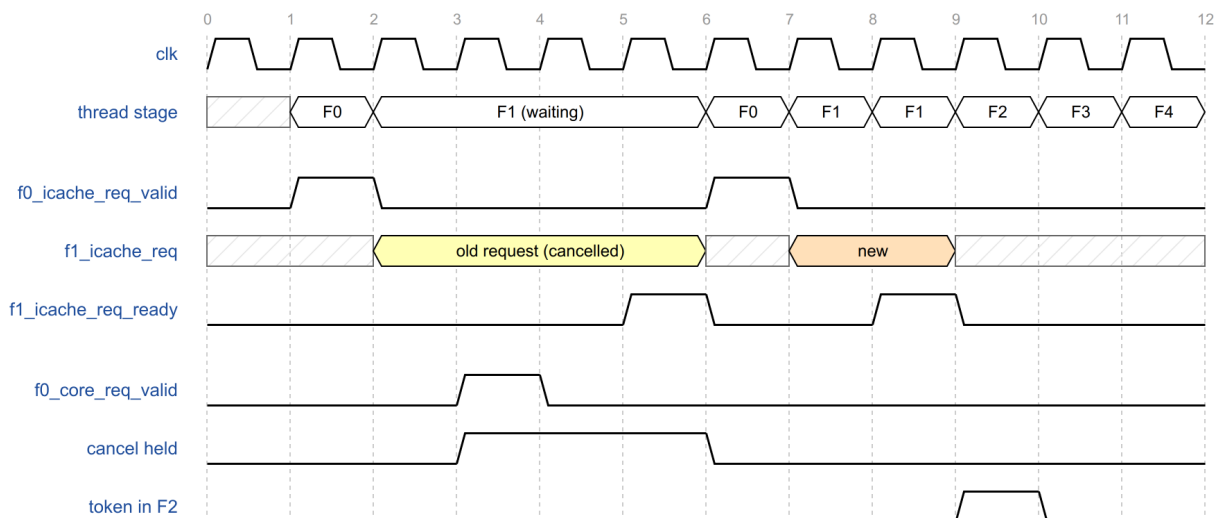


Figure 14. Redirect while a request waits in F1

In Figure 14 the redirect arrives in cycle 3 while the request waits in F1. The cancellation is held until the request is accepted in cycle 5; no token enters F2. The new request is issued in cycle 6 and accepted in cycle 8.

2.2.5. Thread Buffer

The thread buffer (`frontend_thread_buffer`) holds almost all of a thread's Front End state; there is one buffer per thread. This section describes how the buffer stores instructions: two 256-bit entries in a latch register file with a 32-bit, wrap-around read port. It then describes the state kept for each entry, the fetch PC and how it is updated, and the conditions under which the thread requests a new block. It also shows how an entry is reserved and written.

The section then shows the instruction side: how the read pointer extracts one instruction per cycle, including instructions that cross from one entry to the other and the bypass from incoming data. It then covers how the RVC expander turns compressed instructions into 32-bit ones, and which fault, error and replay status is attached to each instruction on its way to the integer pipeline.

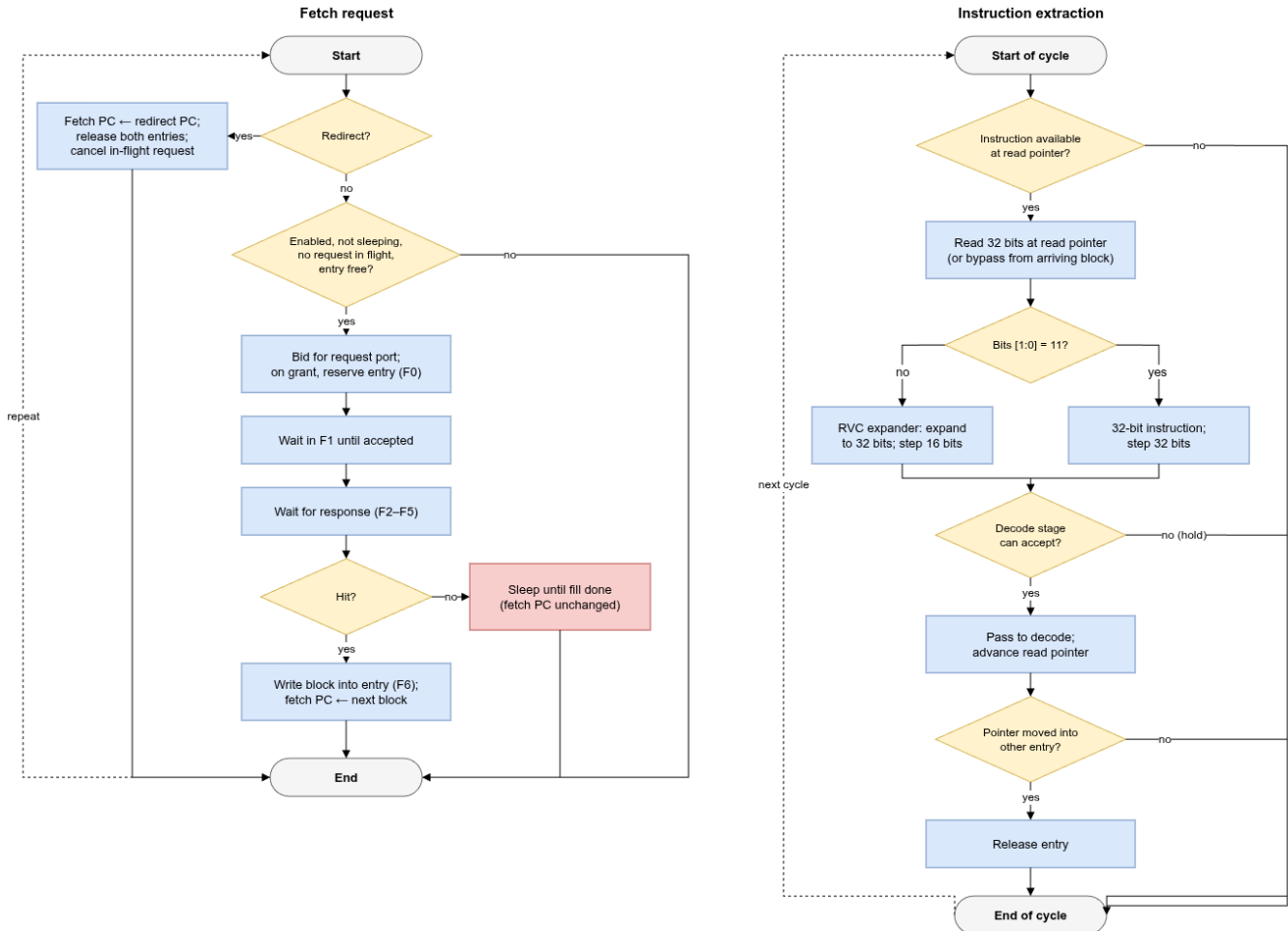


Figure 15. Thread buffer operation

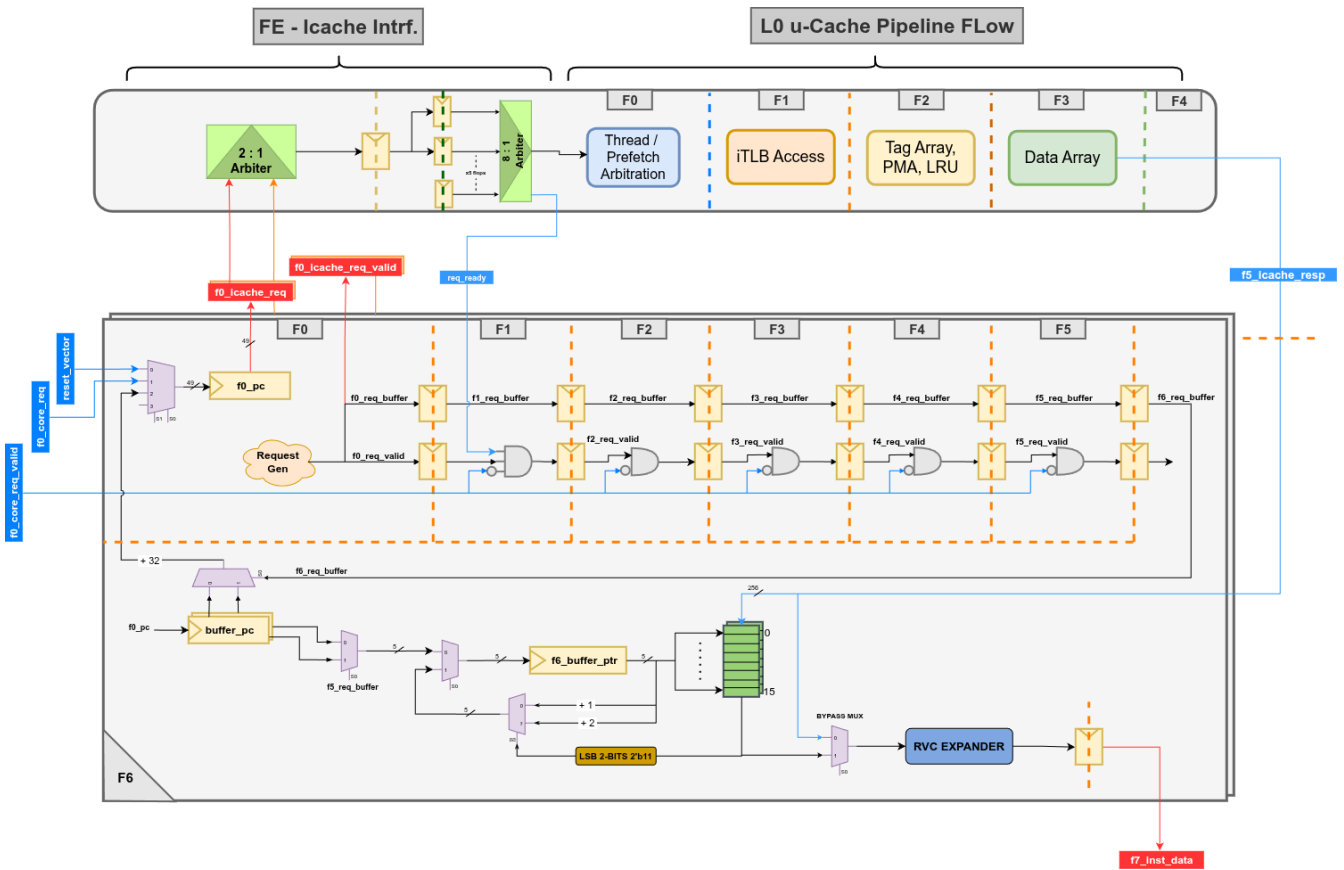


Figure 16. Thread buffer pipeline

2.2.5.1. Buffer Entries

The instruction buffer has two entries of 256 bits (32 bytes). An entry holds one fetch block: eight 32-bit instructions, sixteen compressed instructions, or a combination. Two entries allow the next block to be fetched while the current block is consumed.

The entries are implemented as one latch-based register file with two ports:

- **Write port, 256 bits.** Writes a complete block into one entry, with one write enable per 32-bit word.
- **Read port, 32 bits.** Addressed by the 5-bit read pointer `buffer_ptr`: bit 4 selects the entry, bits 3:0 select a halfword within it. The read returns the 32 bits starting at that halfword.

The read port wraps around: halfword 15 of one entry is followed by halfword 0 of the other. The entries therefore form a ring of 32 halfwords, and a 32-bit instruction that starts in the last halfword of an entry is read in one access.

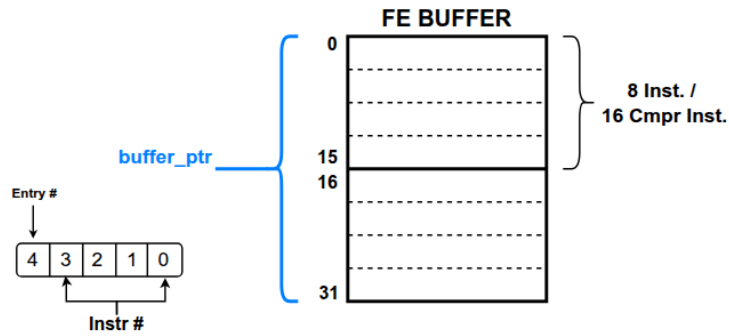


Figure 17. Thread-buffer entries and read pointer

Table 15. Thread-buffer entry state

State	Description
Empty flag	Entry holds no instruction still to be sent. Set for both entries at reset.
Block address	Address of the instruction block held in the entry (<code>buffer_pc</code>). Only the words from this address onward are written, extraction in the entry starts at this address, and it provides the upper PC bits of every instruction read from the entry.
Fault flags	Page fault and access fault of the block.
Error flags	Bus error and ECC error of the block.
Cacheable flag	Block is in a cacheable region. Used for replay.

2.2.5.2. Program Counter

Each thread buffer holds a 49-bit fetch PC, the address of the next block to request. It is updated as listed in Table 16.

Table 16. Fetch PC update

Event	New fetch PC
Reset, or thread enabled	<code>f0_reset_vector</code> , sign-extended.
Block written into an entry	Start of the next aligned 32-byte block.
Redirect (highest priority)	<code>f0_core_req.pc</code> .

Only the first block after a reset or redirect can start within a 32-byte block; all later blocks are aligned. A miss does not change the fetch PC, so the same address is requested again after the fill.

The PC of each extracted instruction is formed from the upper bits of its entry's block address and the halfword position of the read pointer.

2.2.5.3. Request Generation and Entry Fill

A thread issues a request in F0 when all conditions in Table 17 are met.

Table 17. Request conditions

Condition	Reason
Thread enabled, and not in the enable cycle	Reset vector is loaded at the end of the enable cycle.
No redirect in the current cycle	Redirect PC is loaded at the end of the cycle.
Thread not sleeping, or registered fill done in this cycle	After a miss the thread waits for a fill; the request is issued in the cycle the registered fill done arrives.
No request of the thread in F1 to F6	At most one request per thread is in flight, which keeps responses in order and allows ownership by timing.
An entry is empty or is released in the current cycle	The response cannot be stalled and must have a destination.
No halt pending	While a halt is pending the thread sends no request to the L0 micro-cache; it produces a placeholder instruction instead (Section 2.2.10.1).
Not in reset; not executing the debug program buffer	—

On a grant, the thread reserves an entry and records the fetch PC of the request as the entry's block address. If both entries are empty, entry 1 is reserved.

[Table 18](#) shows the entry sequence after a thread is enabled at address A. [m, n] gives the block number in entry 0 and entry 1; x denotes an empty entry.

Table 18. Entry fill sequence

Step	Event	Entries
1	Thread enabled at address A.	[x, x]
2	Block containing A written into entry 1; extraction starts at the halfword of A.	[x, A>>5]
3	Entry 0 empty: the next block is requested when the first request completes.	[(A>>5)+1, A>>5]
4	Last instruction of entry 1 extracted; entry 1 released.	[(A>>5)+1, x]
5	Next block requested into entry 1; extraction continues in entry 0.	[(A>>5)+1, (A>>5)+2]

A block write enables only the words from the word containing the request address to word 7. The register file requires its write-data enables one clock phase before the write; the thread computes them in F5 from its token and `f5_icache_resp_miss`, holds them in a phase latch, and applies the word write enables in F6.

2.2.5.4. Instruction Extraction

When both entries are empty, the read pointer is loaded in F5 with the halfword position of the request address within the reserved entry, so that extraction can start within a block. Afterwards the pointer advances with each extracted instruction.

An instruction is available when the entry it is read from is not empty. A 32-bit instruction that starts in halfword 15 requires both entries to be non-empty; the wrap-around read returns both halves, the pointer moves to halfword 1 of the other entry, and the first entry is released.

If the entry being read is empty but its block is written in the same cycle, the instruction is taken directly from the response data (bypass). The upper half of an instruction that crosses into an entry being written is bypassed in the same way. The first instruction of a block is therefore extracted in the cycle the block is written. The bypass does not check the destination entry of the response; with one request in flight per thread, the only block that can arrive is the one being waited for.

2.2.5.5. RVC Expander

The 32 bits read at the pointer are passed to the RVC expander (`frontend_rvc_expander`), a combinational RV64C decompressor:

- If bits [1:0] are not **11**, the instruction is compressed. It is expanded to its 32-bit equivalent and the pointer advances by 16 bits.
- If bits [1:0] are **11**, the instruction is 32 bits wide. It passes through unchanged and the pointer advances by 32 bits.

All instructions are therefore 32 bits wide after F6. The `rvc` bit of `frontend_core_resp` indicates a compressed instruction; the integer pipeline uses it to advance the PC by 2 instead of 4.

2.2.5.6. Fault and Error Status

Each instruction carries the status of the entries it was read from:

Table 19. Instruction fetch status

Field	Source
<code>pf0, af0</code>	Fault flags of the entry holding the start of the instruction.
<code>pf1, af1</code>	Fault flags of the entry holding the upper half of an instruction that crosses entries; zero otherwise.
<code>bus_error, ecc_error</code>	Error flags of the instruction's entry, or of both entries for an instruction that crosses entries.
<code>replay</code>	Set when the entry's block is not cacheable and <code>f0_core_req.speculative</code> indicates older instructions of the thread in flight. Captured when the instruction is written into the FIFO.

All status fields are forced to zero while the debug program buffer is executed.

2.2.6. Entry Invalidation, Redirect and Thread Start

This section describes the events that discard buffered instructions or restart fetch. [Section 2.2.6.1](#) lists when a buffer entry is released: normally when its last instruction has been extracted, but also on a redirect, a thread disable or entry to debug mode. [Section 2.2.6.2](#) describes a redirect from the integer pipeline: what triggers it, what it clears in each stage, and how long the thread takes to deliver its first target instruction. [Section 2.2.6.3](#) describes how thread enable and disable events

are derived from the enable input, how fetch starts at the reset vector, and what state a thread has after reset.

2.2.6.1. Entry Invalidation

An entry is released when its last instruction leaves F6, or when the whole buffer is flushed ([Table 20](#)).

Table 20. Entry release conditions

Condition	Entries released
32-bit instruction at byte offset 28 of the entry is extracted	This entry
Compressed instruction at byte offset 30 is extracted	This entry
32-bit instruction at byte offset 30 is extracted (upper half in the other entry)	This entry
Redirect (<code>f0_core_req_valid</code>) or thread disable	Both entries
Entry to debug mode	Both entries

A response that reports a fault or error marks the reserved entry non-empty even if no data is written, so that the fault is delivered to the integer pipeline with the next instruction read from that entry.

2.2.6.2. Redirect

The integer pipeline redirects a thread by asserting `f0_core_req_valid` with the new PC in `f0_core_req.pc`. Redirects are issued for branch and jump mispredictions, exceptions and returns from trap, replays, inter-processor interrupts, and entry to or exit from debug mode. [Table 21](#) lists the effect in the redirect cycle.

Table 21. Effect of a redirect

Stage	Effect
F0	Fetch PC loaded with the redirect PC. No request in this cycle. Sleep state cleared.
F1	Request waiting in the F1 I-cache request register marked cancelled.
F2–F5	Tokens invalidated.
F6	No block written. Both entries released.
F7	Instruction discarded.
FIFO	Emptied. The thread is not eligible for scheduling in this cycle.

The target block is requested in the following cycle. Its first instruction reaches ID ten cycles after the redirect ([Figure 18](#)).

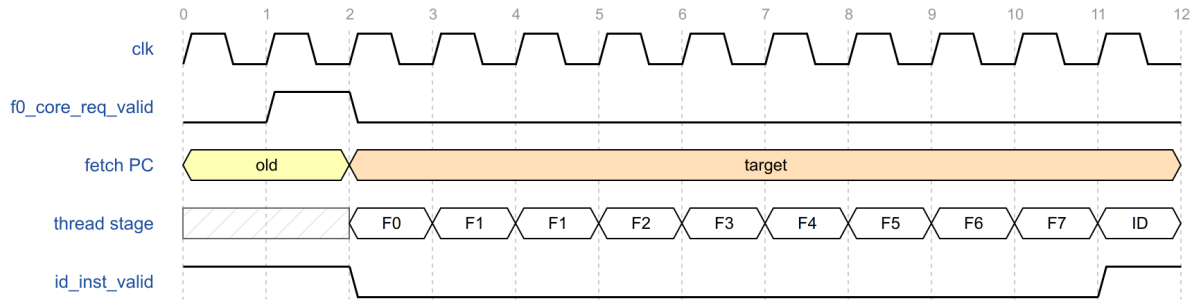


Figure 18. Redirect to first target instruction

2.2.6.3. Thread Enable and Reset Vector

`f0_thread_enabled` is registered on entry to the Front End, and one-cycle enable and disable events are derived from its edges.

- **Enable.** The fetch PC is loaded with `f0_reset_vector` at the end of the enable cycle, and the first request is issued in the next cycle. The first instruction reaches ID eleven cycles after `f0_thread_enabled` rises (Figure 19).
- **Disable.** No further requests are issued, both entries are released and the sleep state is cleared. A fetch already in flight is not cancelled: its response is still written into its reserved entry. The instruction in F7 and the FIFO contents are not cleared. The thread is removed from the scheduler's eligible set one cycle later. Because the scheduler defaults to thread 0, FIFO entries left by a disabled thread 0 can still be presented to ID. The eleven-cycle enable latency applies to a thread whose F7, FIFO and buffer are empty, for example the first enable after reset.

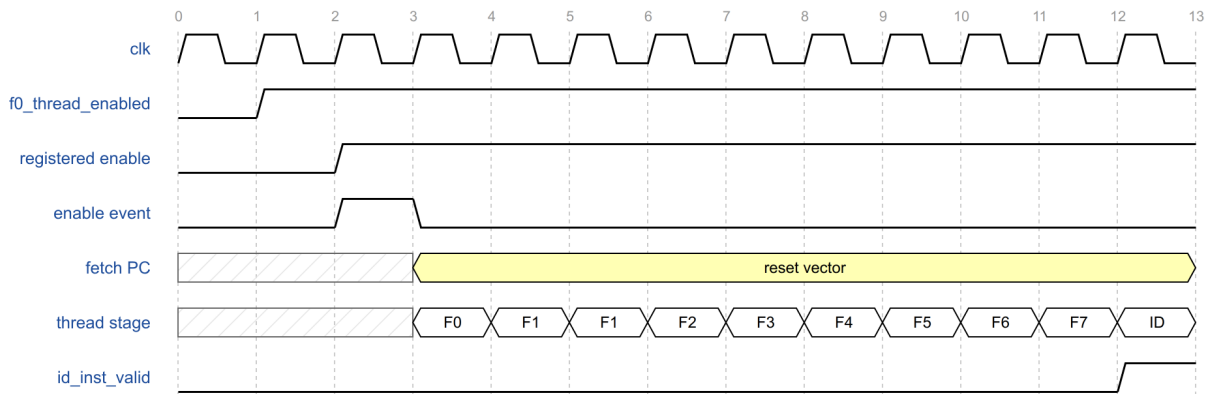


Figure 19. Thread enable to first instruction

After reset, both entries are empty and the fetch PC holds the reset vector.

2.2.7. Decoders

This section describes the integer decoder (`intpipe_decode`) and the VPU decoder (`vpu_decoder`). Both are shared by the two threads and decode in F7, before the instruction is queued. It describes the control words the decoders produce, `minion_control` and `vpu_ctrl_sigs_t`, and how each decoder handles an encoding it does not recognize. It also describes the rule that selects which thread's instruction is decoded in a cycle.

The section also explains how the VPU decode is used once the instruction reaches ID. The Front End does not issue to the VPU itself: the VPU control word goes directly to the VPU. The integer

pipeline issues the instruction to the VPU and uses a summary of the control word for its FP and mask scoreboard checks

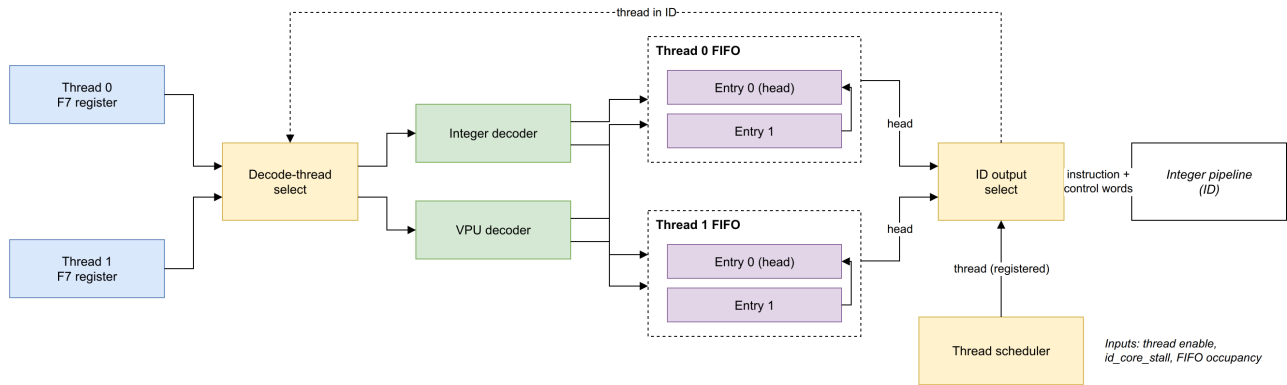


Figure 20. Decoders, instruction FIFOs and thread scheduler

Both decoders are combinational look-up tables over the 32-bit instruction:

- **Integer decoder.** Produces `minion_control` (Table 13). An unlisted encoding is marked not legal, and the integer pipeline raises an illegal-instruction exception.
- **VPU decoder.** Produces `vpu_ctrl_sigs_t` and a flag that identifies VPU instructions. An unlisted encoding is a non-VPU instruction.

A thread is a decode candidate when it has an extracted instruction, its FIFO is not full and it is awake (Section 2.2.9). The decoded thread is selected as follows:

Table 22. Decode thread selection

Candidates	Thread decoded
Thread 0 only	Thread 0
Thread 1 only	Thread 1
Both	Thread not presented in ID in the current cycle
None	Thread not presented in ID in the current cycle

The instruction is written into the FIFO when the selected thread has an extracted instruction and its FIFO is not full; the awake condition affects only the selection.

The Front End does not issue instructions to the VPU. In ID, the VPU control word of the presented instruction (`id_vpu_decoder_sigs`) is sent directly to the VPU, bypassing the integer pipeline. The integer pipeline issues the instruction to the VPU in the same cycle, when the instruction is accepted in ID and decoded as an FP or VPU instruction, together with its thread and that thread's fcsr rounding mode. The integer pipeline also receives a summary of the VPU control word (`id_vpu_core_ctrl`), with the VPU and mask register reads and writes of the instruction, for its FP and mask scoreboard checks.

2.2.8. Instruction FIFO

This section describes the two-entry instruction FIFO that each thread has (uses) between F7 and ID. The FIFO decouples fetch from issue. While ID is stalled or serving the other thread, the thread

keeps extracting and decoding, and the FIFO holds the decoded instructions with their fetch status. The section covers what an entry holds and the pointer scheme that tracks occupancy. It explains why the storage is arranged for timing, as a fixed head register and a holding register, and gives the update rules. It also covers when the VPU control word is valid and how a redirect empties the FIFO.

The FIFO is written when its thread is selected for decode and read when the integer pipeline accepts the thread's instruction (`id_inst_valid`, `id_inst_ready` and `id_inst_thread_id`). It uses 2-bit read and write pointers; it is full when the pointers differ only in the wrap bit and empty when they are equal.

The storage is organized for timing rather than as an indexed array. Entry 0 is always the head, whose content drives the ID outputs; entry 1 is a holding register. The ID multiplexer therefore selects between two fixed registers, one per thread.

Table 23. Instruction FIFO update

Occupancy	Event	Update
Empty	Write	Entry 0 $\leftarrow$ new instruction
One	Write and read	Entry 0 $\leftarrow$ new instruction
One	Write	Entry 1 $\leftarrow$ new instruction
One	Read	None; FIFO becomes empty
Full	Read	Entry 0 $\leftarrow$ entry 1

The VPU control word of an entry is written only for VPU instructions. For other instructions it retains the value of an earlier VPU instruction, so `id_vpu_decoder_sigs` and `id_vpu_core_ctrl` are valid only for VPU instructions.

A redirect empties the thread's FIFO.

2.2.9. Thread Scheduler

The integer pipeline accepts one instruction per cycle from either thread. This section describes how the thread scheduler (`frontend_thread_sched`) chooses which thread's FIFO head is presented to ID in the next cycle. It defines when a thread is awake (enabled and not stalled by the integer pipeline) and when it is ready (it has an instruction and is not being redirected). It gives the round-robin rule between eligible threads, including the default choice when neither thread is eligible. It then describes the handshake with ID: what drives the ID outputs, why the valid signal is not qualified with the stall, and what happens to an instruction that ID does not accept.

Awake

The thread is enabled and `id_core_stall` is deasserted. Awake is registered, so a stall affects scheduling one cycle after it is asserted.

Ready

The thread's FIFO is not empty or is being written in the current cycle, and the thread is not being redirected. Including the write allows an instruction to pass from F7 to ID in consecutive cycles.

A thread that is awake and ready is eligible. The scheduler selects between eligible threads in round-robin order:

Table 24. Thread selection

Thread 0 eligible	Thread 1 eligible	Selected
Yes	No	Thread 0
No	Yes	Thread 1
Yes	Yes	Thread not selected previously
No	No	Thread 0

The previous selection is updated only when at least one thread is eligible. The selection is registered and drives `id_inst_thread_id` in the next cycle.

The head of the selected thread's FIFO drives `id_inst_data`, `id_intpipe_ctrl`, `id_vpu_decoder_sigs` and `id_vpu_core_ctrl`. `id_inst_valid` is asserted when that FIFO is not empty, and the integer pipeline accepts the instruction with `id_inst_ready`. `id_inst_valid` is not qualified with `id_core_stall`; the integer pipeline applies the stall. The thread in ID is reselected every cycle, so an instruction that is not accepted can be replaced by the other thread's instruction in the next cycle; it remains at the head of its FIFO.

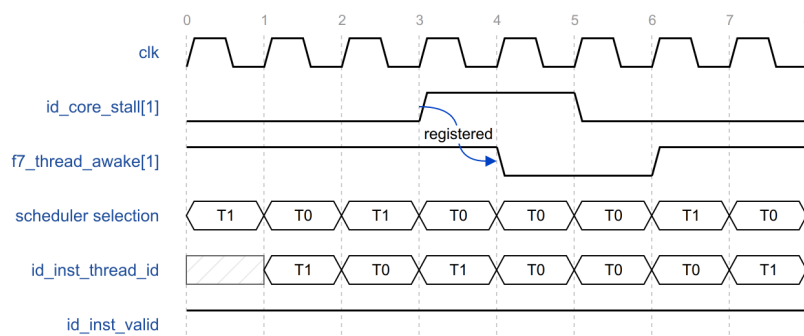


Figure 21. Round-robin issue with a stall

In [Figure 21](#) both threads are eligible and alternate. `id_core_stall` of thread 1 is asserted in cycles 3 and 4, and thread 1 is not awake in cycles 4 and 5. Thread 0 is selected in cycles 3 to 5 and presented in ID in cycles 4 to 6; the selection alternates again from cycle 6.

2.2.10. Halt and Debug Program Buffer

This section describes the Front End's two debug functions. [Section 2.2.10.1](#) covers halt. The integer pipeline takes a halt only on a valid instruction in ID, so while a halt is requested the thread buffer generates a placeholder instruction instead of fetching, and passes it through the normal pipeline to ID.

[Section 2.2.10.2](#) covers the debug program buffer. In debug mode, entry 0 of the halted thread's buffer holds the instructions written by the debugger. The section describes how the buffer is prepared with **EBREAK** on entry to debug mode, how the debugger writes words into it and starts execution, and how a resume redirect ends program-buffer mode. It also shows which write data goes to which word, and why the last word always holds **EBREAK**.

2.2.10.1. Halt

The integer pipeline takes a halt on a valid instruction in ID. While `halt` is asserted for a thread, the thread buffer generates a placeholder instruction instead of fetching:

1. The thread follows the normal request conditions but does not bid for the request port. The request is treated as accepted internally, and its token enters the tracker.
2. The token passes through F1 to F6.
3. In F6 an entry write is forced without a response. The data is the last registered response data; fault and error flags are cleared, and the instruction is treated as 32 bits wide.
4. The placeholder passes through decode and the FIFO to ID, where the integer pipeline takes the halt.

The content of the placeholder is undefined. A pending halt also clears the sleep state.

2.2.10.2. Debug Program Buffer

In debug mode, entry 0 of the halted thread's buffer is used as the program buffer: the debugger writes instructions into it and the thread executes them.

Table 25. Debug program-buffer sequence

Step	Operation
Entry	A redirect with <code>debug_info.halt</code> arrives; the thread stops fetching in the same cycle. In the next cycle both entries are released, the extracted instruction is discarded, and words 0 to 4 of entry 0 are written with <code>EBREAK</code> .
Write	The debugger writes the program buffer through <code>debug_ffb_wdata</code> , <code>debug_ffb_en</code> and <code>debug_ffb_thread_sel</code> (Table 26). The data and enables come together but the data is written into the entry in the cycle after the enables.
Execute	<code>debug_ffb_exec</code> resets the read pointer to halfword 0 of entry 0. In the next cycle entry 0 is marked non-empty and entry 1 empty, and instructions are extracted from word 0.
Exit	A redirect with <code>debug_info.resume</code> ends program-buffer mode; fetch restarts at the redirect PC.

Table 26. Program-buffer word mapping

Word	Write enable	Data
0	<code>debug_ffb_en[0]</code>	<code>debug_ffb_wdata[31:0]</code>
1	<code>debug_ffb_en[1]</code>	<code>debug_ffb_wdata[63:32]</code>
2	<code>debug_ffb_en[2]</code>	<code>debug_ffb_wdata[31:0]</code>
3	<code>debug_ffb_en[3]</code>	<code>debug_ffb_wdata[31:0]</code>
4	—	<code>EBREAK</code>

Word 4 cannot be written by the debugger and always holds `EBREAK`, so execution returns to debug mode at the end of the program buffer. While the program buffer is executed, all fault, error and

replay status is forced to zero. A debug reset while the thread is halted rewrites words 0 to 4 with **EBREAK**.

2.3. Integer Pipeline

2.3.1. Overview

Figure: Integer pipeline block diagram.

2.3.2. Pipeline Stages

Figure: Integer pipeline stages (ID, EX, TAG, MEM, WB).

2.3.3. Interfaces

Table: Integer pipeline interface signals.

2.3.4. Register File and Operand Read

2.3.5. Immediate Generation and ALU

2.3.6. Multiply/Divide Unit

2.3.7. Scoreboards and Hazards

2.3.7.1. Integer Scoreboard

2.3.7.2. FP Scoreboard

2.3.7.3. Mask Scoreboard

2.3.8. Hart Interleaving and Stalls

2.3.9. Branches, Redirects and Replay

2.3.10. Interface to the Data Cache

2.3.11. Interface to the VPU

2.3.12. CSRs and Privilege Modes

Table: CSR groups.

2.3.13. Exceptions and Interrupts

2.3.14. Debug and Breakpoints

2.3.15. Performance Counters

2.3.16. Verification Notes

2.4. Data Cache

2.4.1. Overview

Figure: Data cache block diagram.

2.4.2. Organization and Operating Modes

Table: Data cache operating modes.

2.4.3. Interfaces

Table: Data cache interface signals.

2.4.4. Request Pipeline

Figure: Data cache request pipeline.

2.4.5. Address Translation and PMA Checks

2.4.6. Tag, Data and LRU Arrays

2.4.7. Miss Handling

2.4.8. Replay Queue

2.4.9. Writeback and Store Merge

2.4.10. Atomic Operations

2.4.11. Cache-Management Operations

2.4.12. Scratchpad Control

2.4.13. Tensor Load and Reduce

2.4.14. Interface to the Neighborhood

2.4.15. Errors

2.4.16. Performance

2.4.17. Verification Notes

2.5. Vector Processing Unit

2.5.1. Overview

Figure: VPU block diagram.

2.5.2. Interfaces

Table: VPU interface signals.

2.5.3. Control and Micro-Instruction Decode

2.5.4. Register Files

2.5.5. Lanes

Figure: VPU lane.

2.5.6. Mask Unit

2.5.7. Bypass Network

2.5.8. Shift and Swizzle Unit

2.5.9. TxFMA Unit

2.5.10. Integer Multiply-Accumulate Unit

2.5.11. Transcendental Unit

2.5.12. Tensor Operations

2.5.12.1. Tensor FMA

2.5.12.2. Tensor Quantization

2.5.12.3. Tensor Reduce

2.5.13. Interfaces to the Integer Pipeline and Data Cache

2.5.14. Performance

2.5.15. Verification Notes

3. Neighborhood

3.1. Neighborhood Overview

3.1.1. Partitioning

Figure: Neighborhood block diagram.

3.1.2. Shared Resources and Core Connections

3.1.3. Voltage-Domain Crossings

3.1.4. External Interfaces

Table: Neighborhood interface groups.

3.2. Front End – I-Cache Interface

3.2.1. Overview

Figure: Front End to I-cache connection within a Neighborhood.

3.2.2. Interface Signals

Table: Front End – I-cache interface signals.

3.2.3. Request Arbitration

3.2.4. Request and Response Timing

Figure: Timing of a request and its hit response.

3.2.5. Miss, Sleep and Fill-Done Protocol

Figure: Timing of a miss, fill and retry.

3.2.6. Fault and Error Reporting

3.2.7. Livelock Avoidance

3.2.8. Known Limitations

3.3. L0 Micro-Cache

3.3.1. Overview

Table: L0 micro-cache characteristics.

3.3.2. Internal Blocks

Figure: L0 micro-cache block diagram.

3.3.3. Pipeline Stages

Figure: L0 micro-cache pipeline.

3.3.4. Instruction TLB

3.3.5. Micro-Tag and Micro-Data Arrays

3.3.6. PMA Check

3.3.7. Replacement

3.3.8. Miss Handling

3.3.8.1. TLB Miss

3.3.8.2. Tag Miss

3.3.8.3. Fill and Thread Wake-Up

3.3.9. L1 Bypass

3.3.10. Software Prefetch

Figure: Prefetch state machine.

3.4. L1 Instruction Cache

3.4.1. Overview

Figure: L1 instruction cache block diagram.

3.4.2. Organization

3.4.3. Tag, LRU and Data Arrays

3.4.4. External Data RAM

3.4.5. Request Arbitration Between L0 Micro-Caches

3.4.6. Miss Path

3.4.7. Errors and ECC

3.5. Shared Page-Table Walker

3.5.1. Clients and Arbitration

3.5.2. Walk Sequence and Page Sizes

Figure: Page-table walk state machine.

3.5.3. PTE Cache and Invalidation

3.6. Memory Request and Response Path

3.6.1. Miss, Evict and Fill FIFOs

Figure: Neighborhood memory request and response path.

3.6.2. Request Arbitration

3.6.3. Response Routing and Backpressure

3.7. Tensor Support

3.7.1. Cooperative Tensor Load

3.7.2. Tensor Store Buffer

3.7.3. TBOX Router

3.8. Synchronization, Messaging, Registers and PMU

3.8.1. Fast Local Barrier

3.8.2. Local Message Network

3.8.3. IPI and FCC Paths

3.8.4. Neighborhood Registers and APB Access

3.8.5. Debug Channel

3.8.6. Performance Monitoring Unit