META HTP

GP Technical Reference Manual - Architecture Overview

Filename	:	Meta HTP.GP Technical Reference Manual - Architecture Overview.doc
Version	:	2.1.314 External Issue HTP.
Issue Date	:	30 Apr 2013
Author	:	Imagination Technologies Limited

Contents

1. Introduction

This document provides an overview of the main features and detailed reference information for the Meta HTP hardware multi-threaded processor core.

A growing number of variants of the Meta core exist with differing sets of features. Some sections of this document describe the Meta processor core in general terms. It should be noted that not all features described are included on every variant. A configuration specification document is supplied with each core to clarify the exact feature set of that core.

The general purpose Meta instruction set is described fully in the 'Meta GP Technical Reference Manual - Instruction Set' document which should be read in conjunction with this document. For further information about DSP and FPU architectural features and instruction sets please refer to the relevant DSP or FPU Technical Reference Manual supplied.

1.1. Meta core architecture highlights

The Meta core is a multi-threaded core with digital signal processing (DSP) and floating point (FPU) capabilities for use in systems such as audio signal processing, digital wireless, mobile, set-top box, automotive and other applications. The Meta's real-time, Linux-capable architecture can execute multiple general purpose and DSP tasks on the same core without any cross-task interference.

It is a high-performance, low-power, modular device enabling extensive customisation. The major functional units such as the number of threads, caches, DSP and FPU resources are variable, which has allowed a wide range of cores in the family to be produced at various levels of performance, power and die size, which enables the development of optimum solutions in silicon.

Item	Features
CPU	RISC, Load/Store architecture.
	Three-operand register based.
	One to four independent hardware threads.
	Threads share resources such as register execution units and coprocessor ports and have some separate resources such as program counters and other 'local' registers.
	Instructions per cycle: Mostly single cycle operations and register-to-register operations. Simultaneous multi-threading - allows instructions from more than one thread to be issued on the same clock cycle.
	One memory read or write per clock.
	8-13 stage pipeline.
	Branch prediction and speculative execution.
Data types and registers	Data Unit - variable number of per-thread and global execution registers.
	Address Unit - variable number of per-thread and global execution registers.
Instruction Set	RISC 32-bit instructions.
	Some single instruction/multiple data (SIMD) instructions.
	MiniM 16-bit instructions for reduced code size.
Floating point support	IEEE compliant single-precision and double-precision floating-point arithmetic.
	Some non-IEEE compliant operations suited to DSP applications.
Caches	Separate instruction and data caches.

Item	Features
	2x64-bit wide interface for parallel code and data accesses per cycle .
	Instruction cache - non-blocking, 4-way set associative or 2-way skewed associative cache up to 64Kbytes size.
	Data cache - non-blocking, 4-way set associative or 2-way skewed associative cache up to 64Kbytes size.
	Flexible Partitioning between threads.
Memory subsystem	External Bus master interface width 32, 64 or 128-bit IMGBUS 3.0 compliant.
	Write combining and burst gathering for optimised SDRAM/DDR utilisation.
	FIFO pipelined memory read queue.
Bandwidths	Main Caches 64-bit instruction cache instruction fetch per clock with per thread pre-fetch buffering. 64-bit core code memory fetch per clock. 64-bit data cache read per clock. 64-bit data cache write per clock. .
	Internal register files Address unit - 2x32 bit reads, 1x32 bit write per clock. Data unit - 2x32 bit reads, 1x32 bit write per clock.
	Up to eight read or write 64-bit coprocessor ports.
Memory Management Unit	Variable 4K to 1Mbyte page-based address mapping, access, and cache control.
	Optimised for Linux
	4GByte Virtual Memory Space ~2GByte per-thread and ~2GByte global address spaces.
	Space for an extensible set of memory system bus transactions.
	Internal support for cache flush, cache on/off, and bus lock/unlock.

2. Meta core architecture overview

The Meta core supports one to four independent hardware threads which typically work in parallel on independent activities. A typical implementation might have four threads, two DSP and two general purpose. A lightweight implementation might have two threads, one general purpose and one DSP.

The Meta core can perform a number of real-time and non-real-time, GP, FPU and DSP processing tasks concurrently in a single core where traditional solutions would require a multi-processor system to achieve similar performance.

Tasks may also be automatically split between threads in an SMP operating system environment.

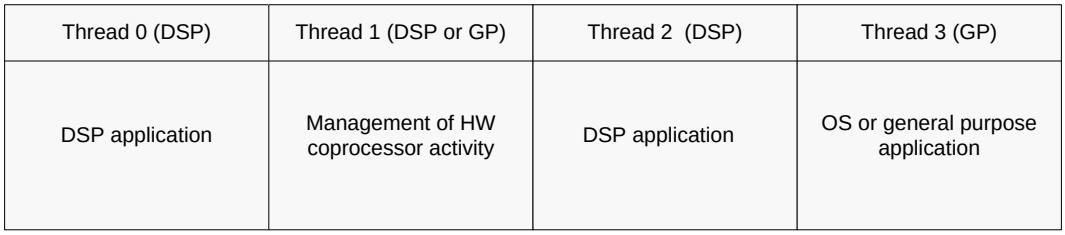

Figure 1 Example of thread employment on a Meta SoC

Unlike traditional processors that are underused due to multi-cycle memory latencies or waste time performing context switches in software, the Meta core supports multiple threads in hardware, with each thread being an instantiation of the processor.

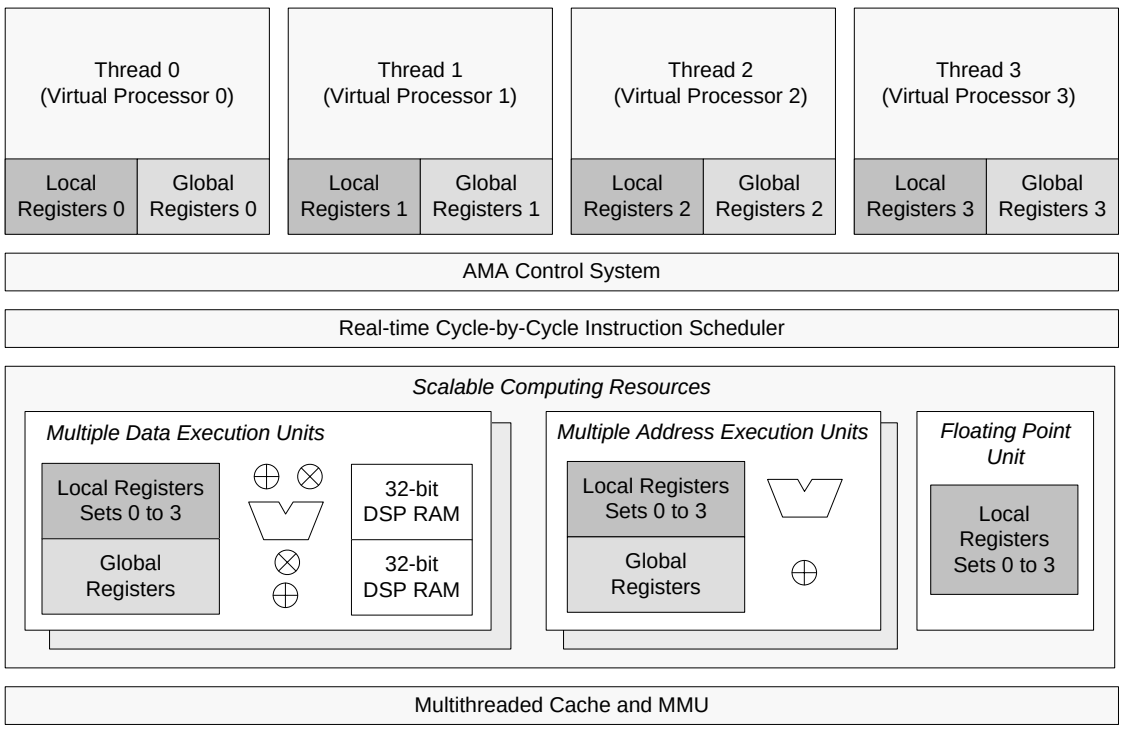

Figure 2 Basic architecture and operation

Threads are effectively separate processors that share the processor's core resources such as register execution units (ALUs, multiplier, accumulator etc.) and coprocessor ports, but have some separate resources such as read/write ports.

Although the processing resources are shared, to accommodate multiple thread contexts, each execution unit holds a local register state, an execution pipeline and a program counter (PC) for each thread. A separate control unit holds mode bits and control registers for each thread.

DSP RAM is an optional global resource shared between threads specifically for use in the extended DSP instruction set. For more information see the Meta DSP Technical Reference Manual.

A fine-grained instruction scheduler switches between the thread contexts on a cycle-by-cycle basis. This scheduler requires that all instructions are atomic and will complete inside a known number of cycles).

2.1. Managing resources for multiple threads

With Meta multi-threading, all threads operate in a parallel/overlapped manner and can switch contexts in response to real-time events without software overhead. In the event of a condition that can potentially cause a stall cycle (e.g. a cache miss), the Meta core automatically starts executing the next thread. When a specific thread must run, the Meta core provides a number of features including cache line locking, cache pre-fetching to control memory stalls, and data address pre-issue to avoid pipeline delays.

Unlike multi-processor systems, where care must be taken early on to partition tasks between processors, the Meta core allows developers to regard the code on each thread as if it is the only code present. They can develop real-time applications in isolation, and later run them in parallel on separate threads. The intricacies of multi-threading are automatically handled by the Meta hardware and software development tools.

2.1.1. Inter-thread communications

A thread encompasses all the features that a conventional processor provides for the execution of a task, including independent support for interrupt handling, software scheduling and privilege protection. Threads are equivalent to multiple independent processors operating together.

There are times when threads are required to interact. The Meta core provides several ways of achieving this:

- Hardware events; (triggers) generated by one thread and sent to another.
- Kick hardware; where software-generated events are stored for later processing by a thread.
- Signals; a software extension of the hardware kick system. Up to 32 independent events with corresponding handlers may be established for each thread by setting a bit and then sending a kick.
- Shared memory areas into which threads may read/write via an agreed protocol.
- Thread synchronization; using memory bus interlocks that allow atomic updates to key shared memory locations to be used for synchronization
- Cyclic command buffer; a shared memory cyclic buffer exploiting the kick system, in which command descriptions are placed and then corresponding kicks are delivered to the server thread.

By combining the methods described above, a unified system of multi-level thread interaction can be created, using the hardware scheduler and simple run-time code, without the need for an operating system.

2.1.2. Instruction scheduler

The instruction scheduler manages multiple threads by extracting a list of required resources from the next pending instruction for each thread. Resource requirements are matched to resource availability via an interlocking process that yields a set of instructions that can be issued. Over fifty internal resources are considered by the scheduler to determine whether a thread can run on the next cycle. From this set of possible instructions, one instruction is chosen to issue according a variable priority scheduler.

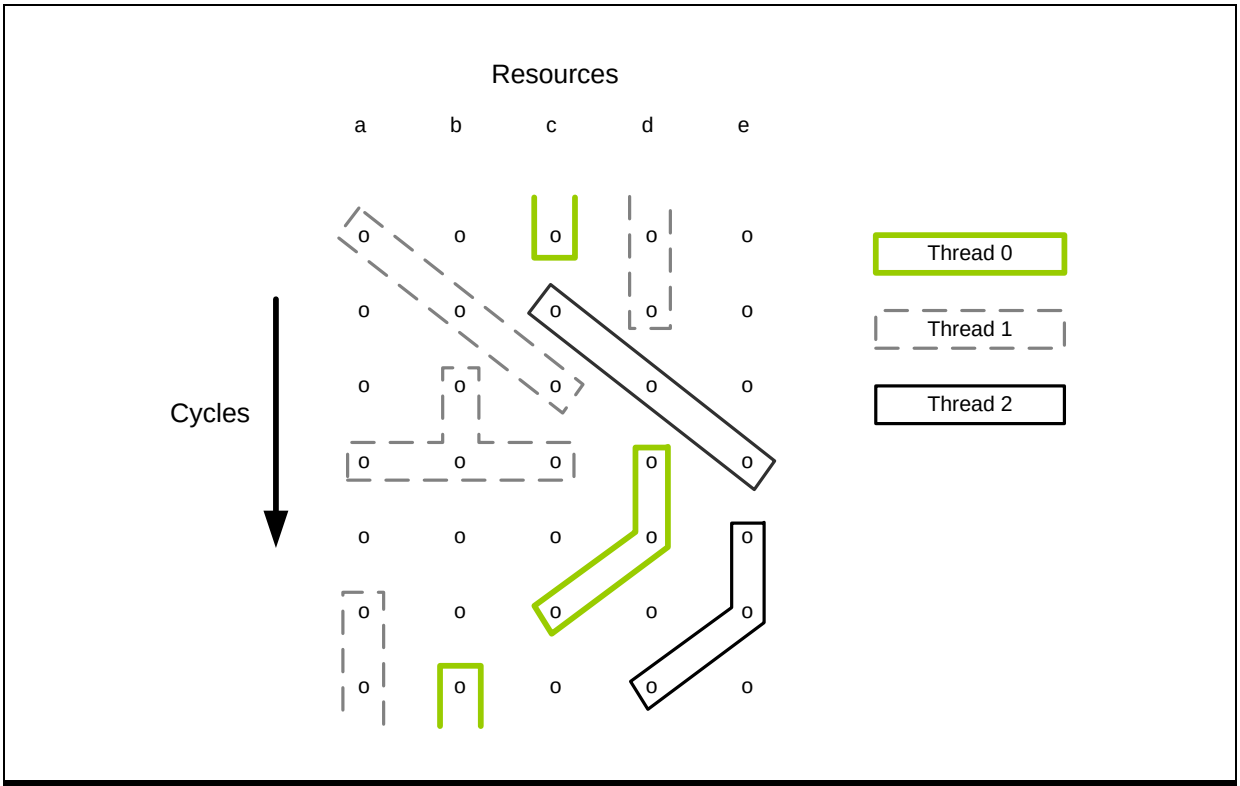

Figure 3 Thread resource scheduling

Each thread can use different processor resources at the same time, or one thread can use all of the processors resources as shown above. This means key algorithms such as Fast Fourier Transforms are performed extremely quickly and mundane actions such as data movement are done with maximum efficiency.

To obtain instructions to present to the instruction scheduler each thread includes an independent instruction fetch engine that can obtain instruction data from a core memory or via the instruction cache.

To provide enhanced performance the threads' fetch engines may make predictions regarding the program flow (dynamic branch prediction) and may fetch and/or execute instructions based upon those predictions (speculative execution).

Notably, sections of memory where code and data are intermingled the fetch prediction is blind to the distinction and may make predictions based upon data items instead of regular instructions.

2.1.3. AMA™

Meta's patented AMA (Automatic MIPS Allocation) provides automatic resource management in hardware, ensuring that each thread of execution gets the MIPS it needs and has the required response time.

2.1.4. Memory reads

The Meta core manages memory reads in the same way as a typical RISC CPU. Interlocks exist so that if a load from memory request is missed, the affected thread is descheduled until the missed data returns. All other threads continue to operate as normal.

2.1.5. Privilege/Lock

Privilege

At times threads may need to be protected from external interference, for example to implement operating system protection. In the Meta core all shared resources have a privilege mode flag to indicate their availability. The memory sub-system also supports the concept of privilege.

For core registers, privilege violations are evaluated before the instruction is scheduled and violations may invoke background or interrupt triggers.

A thread may optionally change privilege when it enters an interrupt handler. At all other times threads remain at their original privilege level.

Lock - thread exclusion

Similarly, it is necessary to support mutual exclusion between threads for some critical operations to be implemented.

There are three levels of thread exclusion currently supported by a special lockinstruction:

Level 0 - no exclusion

All threads are free to execute all instructions. This is the default state for all threads.

Level 1 - voluntary exclusion

This only affects threads that attempt to execute the voluntary lock instruction, LOCK1. Once executed on a particular thread, all other threads that attempt to execute LOCK1 will be prevented from acquiring the voluntary lock and will not be allowed to execute beyond that point until the retaining thread executes the voluntary unlock instruction.

Level 2 - global exclusion

Requests the global exclusion level and prevents all other threads from executing any other instructions until the retaining thread executes the global unlock instruction.

Use of this feature must be limited to a sequence of a few instructions if the real time performance of the other threads is not to be adversely effected. Generally the only reason why this sort of exclusion is required is because some state that would effect the execution of the other threads is being modified and may become temporarily invalid for the other threads to use during the update concerned.

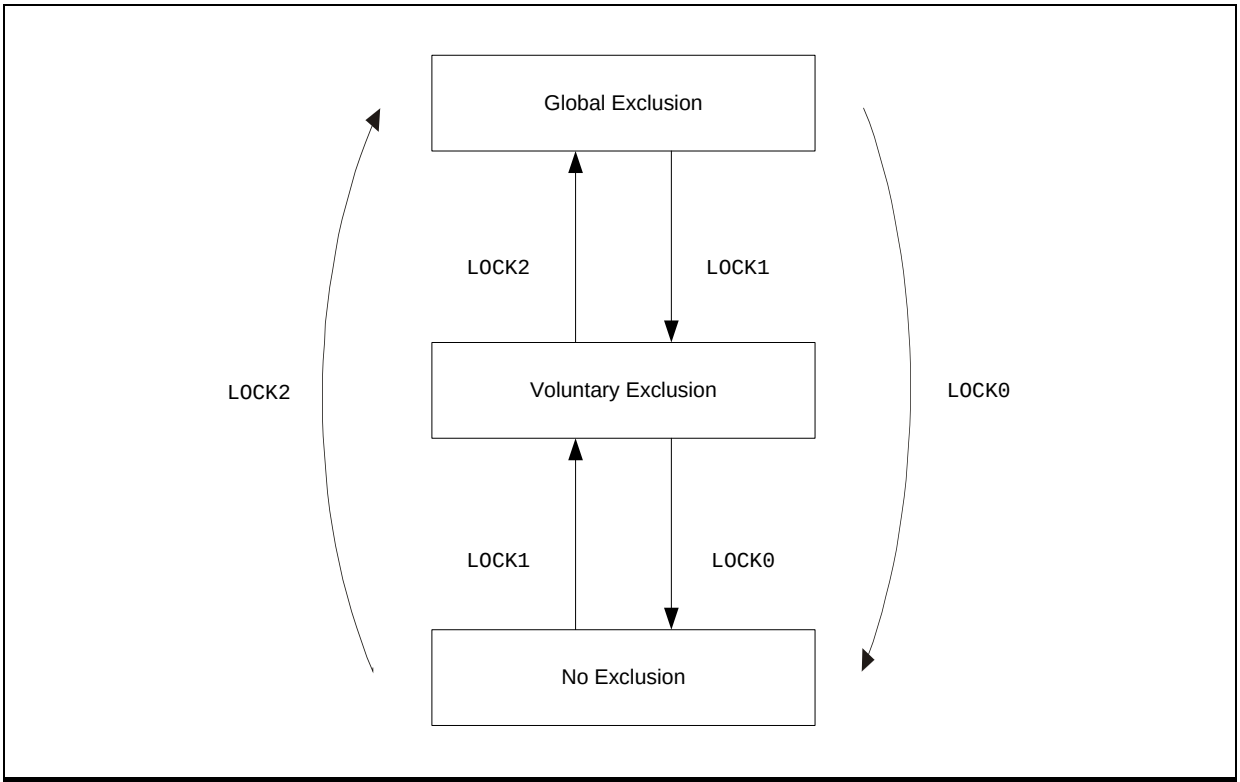

Figure 4 Movement between thread exclusion levels

2.2. Interfaces to the Meta core

The block diagram below shows the interfaces to a Meta core when implemented in a typical Meta SoC device. Brief descriptions are given in this section of the various interfaces. More detailed descriptions are given later in the document.

2.2.1. Coprocessor interface module

Up to eight read and/or write coprocessor interfaces can be implemented.

The coprocessor interface module lets data be transferred to and from any application specific hardware modules, for example real-time data feeds such as digital audio.

Coprocessor ports are usually synchronous interfaces carrying up to 64-bits of data. How that data is interpreted depends entirely on the coprocessor. In general there is no rule for what provokes the arrival of data on a coprocessor read port. It may be the result of a previous coprocessor write operation or the occurrence of an external event, but it is totally specific to the coprocessor concerned.

A coprocessor read or write operation will only be executed when it can immediately complete, until this state is reached, the thread concerned is descheduled. In a multi-threaded core another thread can be scheduled to execute during this time, in a single-threaded core the thread is blocked until the operation can complete.

2.2.2. Reset

After the rising edge of a reset the Meta core can boot from a fixed internal location or externally via the Slave or Debug interfaces.

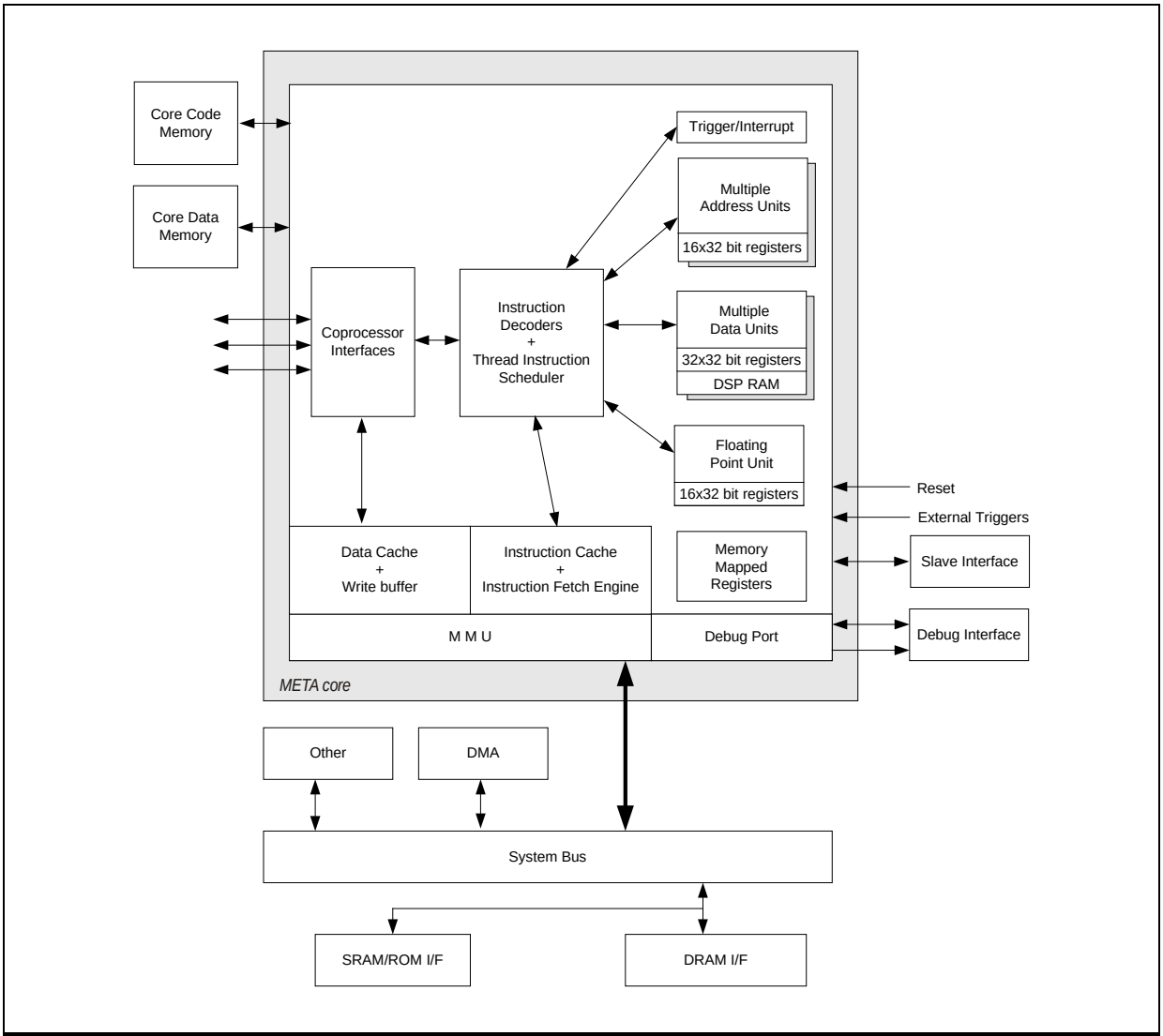

Figure 5 Core interfaces in a typical Meta device

Note: The number of registers shown is for a typical implementation and is variable, please refer to the Core.Configuration Specification document supplied for details.

2.2.3. External triggers

Triggers provide a method of interlocking with, or interrupting, the execution of threads in the Meta core. Up to 128 external system triggers may be input from outside the core and configured in the core as edge sensitive or level sensitive signals.

2.2.4. Slave interface

The Slave interface allows the Meta core to be booted, configured and controlled by a Host processor.

This slave interface allows “Kicks” to be addressed externally to individual counters that are specific to interrupt or background processing levels of each thread.

2.2.5. System bus master

The physical memory map is implemented for the Meta core via the System Bus port. This port can carry a number of simultaneous transactions for each thread, allowing independent operation of threads from memory mapped hardware which might have different response times. Requests to the system bus are made via the caches and MMU:

Caches

The Meta core interfaces directly to code and data caches that support its full read/write code/data operating bandwidth. Using this interface the core can issue both a data read/write operation and a code read operation in a single cycle with the intention of moving data into and out of the corresponding caches as fast as possible in parallel.

MMU

This interface allows the data and code caches to obtain access to external memory in a common unified memory map controlled by the MMU. The MMU is responsible for translating the virtual addresses used by Meta threads and coprocessor hardware into physical memory addresses implemented via the System Bus.

2.2.6. Debug interface

This interface allows an external debug host to indirectly issue reads or writes to the logical address space of any thread. This allows all features of the core to be controlled or monitored. This interface is accessed via a JTAG TAP controller positioned at the top level of the customer SoC.

2.2.7. Core memory ports

These interfaces allow arbitrarily sized closely coupled instruction memory and data memory to be attached to the Meta core. These memories would typically be used for key DSP algorithms depending on the application.

2.3. Execution units

The Meta core's logic is built around execution units which help to support multiple thread contexts and to improve the load balancing of those contexts. Execution units hold a local register state and an execution pipeline for each thread. Different types of execution unit have different pipeline logic and a different set of registers. A separate control unit holds mode bits and control registers for each thread, and a separate unit holds a program counter (PC) for each thread.

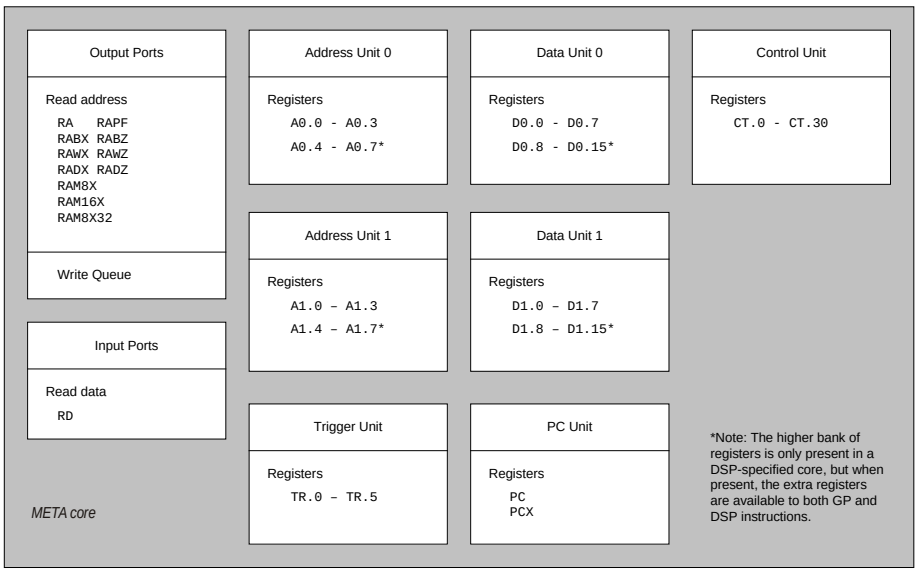

Figure 6 Per-thread resources

Note: The number of registers shown is for a typical implementation and is variable, please refer to the Core.Configuration Specification document supplied for details.

Register execution units

- Data unit
- Address unit
- PC unit

Non-execution units

- Trigger unit
- Control unit
- Input/output ports
- Coprocessor ports

The use of units allows a large total number of registers to be incorporated into the design. It is possible for instructions to make use of more than one unit at a time to enable parallel execution and improve performance.

The number of arithmetic execution units is two address units and two data units. A DSP enabled arithmetic unit has extra registers, but when these registers are present they are available to both DSP and GP instructions. This does not apply to accumulator registers which are DSP only.

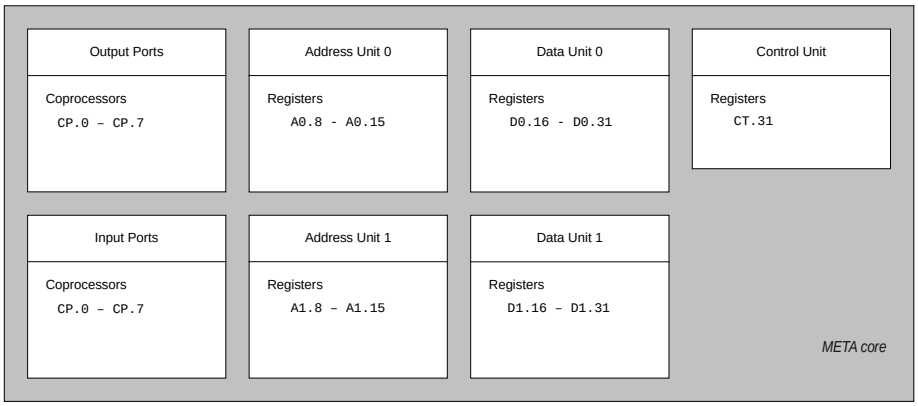

Figure 7 Global resources

Notes:

1 The number of registers shown is for a typical implementation and is variable, please refer to the Core.Configuration Specification document supplied for details.

2 For single-threaded cores, resources shown as global above may not be present.

Internal control registers can be referred to by their register number or their alias (e.g. CT.0 =TXENABLE) as defined in metag.inc in the Meta toolkit. The registers shown below are for a four-threaded core, not all registers may be available depending on the specific implementation.

GP control unit registers		Trigger unit registers	
TXENABLE	CT.0	TXSTAT	TR.0
TXSTATUS	CT.2	TXMASK	TR.1
TXRPT	CT.3	TXSTATI	TR.2
TXTIMER	CT.4	TXMASKI	TR.3
Reserved	CT.11	TXPOLL	TR.4
TXTIMERI	CT.13	TXPOLLI	TR.6
TXCATCH0-3	CT.16-19		
TXDEFR*	CT.20		
Reserved	CT.22		
Reserved	CT.23		
Reserved	CT.24-27		
TXDIVTIME	CT.28		
TXPRIVEXT	CT.29		
TXTACTCYC	CT.30		
TXIDLECYC	CT.31		
*HTP onwards			

2.3.1. Data unit

The data units contain a 32-bit register file with a maximum of 16 registers (0-15) per thread and 16 global registers (16-31). These registers are internally linked to an ALU, which can perform signed add/sub, logical-arithmetic-left/right shifts, logical and/or/xor, address operations, and 16/32-bit multiplies. The data unit's ALU can generate conditions that affect the future execution of the instruction stream.

The block diagram in Figure 8 shows, at the top of the diagram, the register file that contains the main register storage for the unit. This has two read ports (rop1 and rop2) and two write ports (int and ext). The read ports allow two register operands to be read from the register file per cycle, while the two write ports allow writes from two separate pipeline phases to be committed to the register file per cycle.

This is followed by a number of multiplexed data paths that select which parameters will be loaded into the Op1 and Op2 registers in each of the three sub-pipelines (logicals, add/sub and multiply). There are various circumstances under which more than one sub-pipeline may be active at the same time - for example the add/sub pipeline may take two cycles to obtain the two results of an FFT butterfly add/sub during which other instructions can use a different sub-pipeline.

The pipelined data path includes some pipeline re-circulation capabilities which can enable continuous throughput for unit-internal (UI) operations.

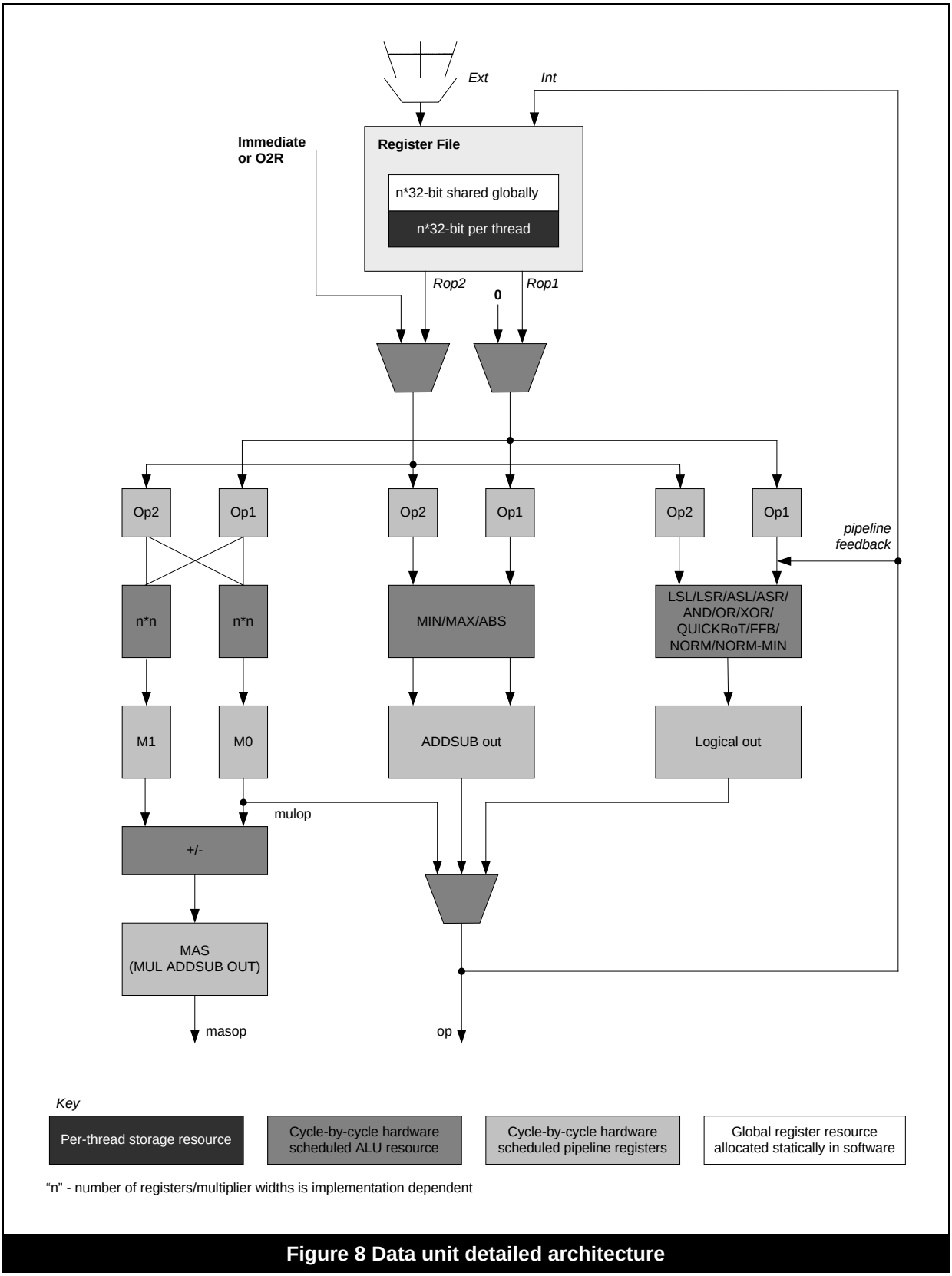

Figure 8 Data unit detailed architecture

2.3.2. Address unit

Address units consist of a smaller group of registers with simpler functionality specifically targeted towards memory address generation. Each address unit holds eight registers that are private to each thread (0-7) and eight global registers (8-15).

As shown in Figure 9, the register files in each address unit have two read ports (rop1 and rop2) along with two write ports (int and ext). The two read ports allow two register operands to be read from the register file per cycle, while the two write ports allow writes from two separate pipeline phases to be committed to the register file per cycle.

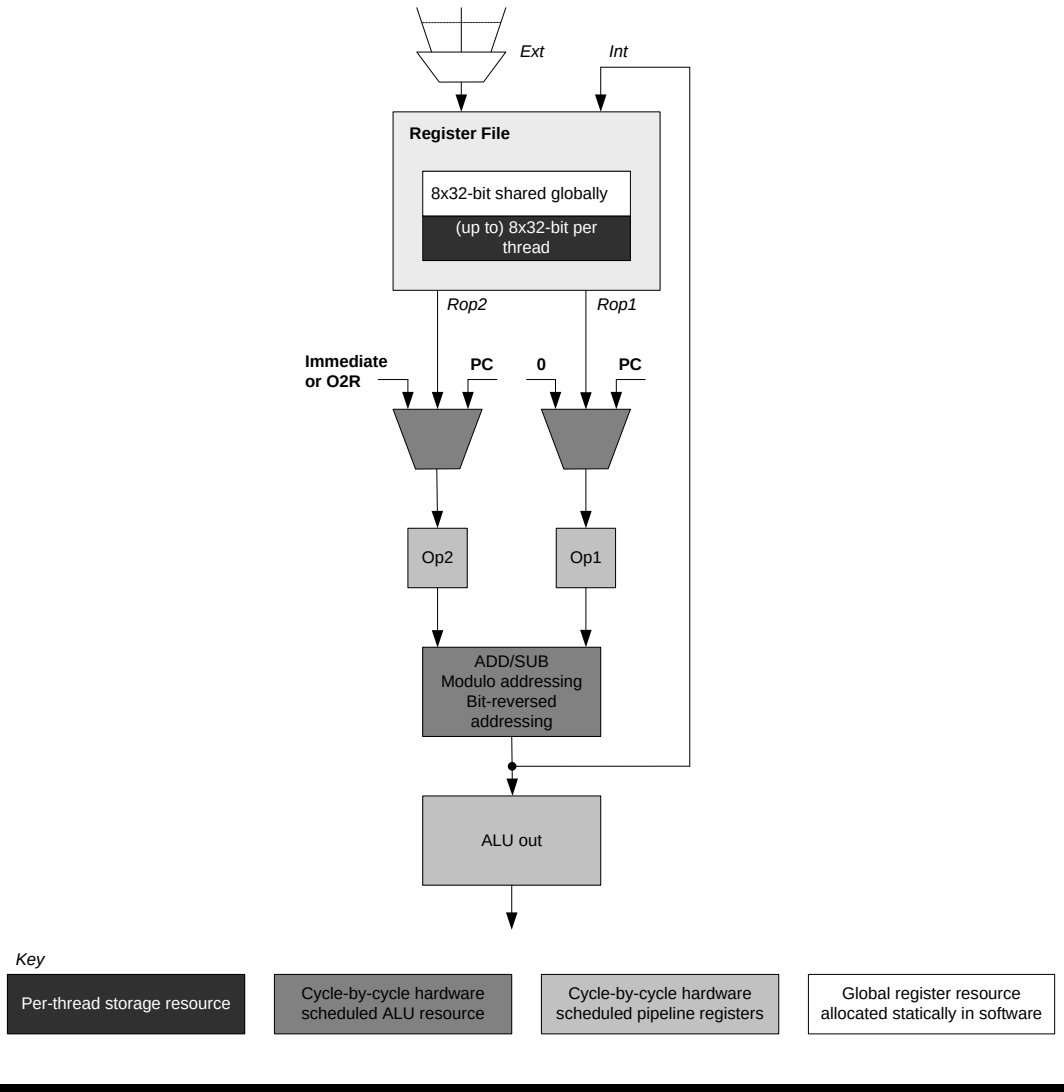

Figure 9 Address unit detailed architecture

The value of the current PC address can be used as either of the source parameters. This allows calculations relative to the address of the current instruction to be initiated quickly without the need to explicitly transfer a value from the PC unit. The value of the PC cannot be derived directly in this way in other units, however the value derived from registers in the address units may be delivered into any other unit.

To support the requirements for DSP-style functionality, address units enable the issue of a load or store in parallel with the operation of the main DSP execution units.

Addressing modes

As well as normal addressing, the address units have some extra DSP specific functionality; modulo addressing and bit reversed addressing these are described further in the DSP Technical Reference Manual.

2.3.3. PC unit

The PC unit holds two program counter registers per thread, along with a simple functional unit that is used for the majority of the system's program flow manipulations such as branches and hardware loops.

During the execution of normal background code, PC holds the execution address for the current thread, and PCX holds the address from which the next interrupt on this thread will execute. When an interrupt occurs, the contents of PC and PCX swap. PC holds the execution address for the executing interrupt handler, while PCX holds the address from which code execution will resume when the interrupt completes. The swapping of PC and PCX happens transparently when interrupt level is entered or exited.

Branches, jumps and calls must use the address for the first instruction to be executed, as no further processing is applied to this value before it is copied into a thread's PC register.

2.3.4. Control unit

The control unit is a simple unit that only contains a register file and has no associated ALU.

The registers hold all of the control state that cannot be put into memory mapped I/O space, i.e. all the control registers that are needed in the core itself such as individual thread on/off controls, DSP mode switches, hardware loop controls and repeat counters.

Internally, each thread has a direct access to its own block of registers (CT.0 to CT.31). Some of these registers require the thread to be privileged before it can write to them (unprivileged reads are always allowed).

To access a register in another thread's block of registers, a memory-mapped access must be performed, so all control unit registers of all threads can be globally accessed via a memory-mapped region allocated to these registers.

2.3.5. Trigger unit

The trigger unit provides a mechanism for detecting and synchronizing with various types of system event. It holds six registers which control routing of internally generated and externally supplied triggers, and which allow the original source of a processor trigger to be located.

The trigger unit provides independent systems for two levels of event handling; background and interrupt. Background control provides voluntary synchronization with zero overheads, whilst interrupt control is based on a conventional model with overheads.

See section 2.4 Exceptions, Triggers and Kicks for more details of triggers.

2.3.6. Input/output ports

The Meta core has several memory ports that connect to internal and external data sources. DSP performance is assisted by using separate instruction and data caches, or on-chip RAMs, to reduce the work of the memory interface. For example, when a program is operating in a tight core loop, all requests can be serviced by the instruction cache or on-chip RAMs so there is no instruction fetch activity on the memory bus.

Interaction of memory ports

Data can be retrieved via the load port or read port to suit the application.

Loads

Loads get data from memory and put it in the register concerned.

Reads

Reads target the read pipeline, having separate read and return instruction issues as described below. Read ports are most useful in DSP applications.

Memory load ports, read ports, and write ports are linked to provide coherency and efficient implementation. Loads, reads and writes may be combined in any order.

Pipelined memory

To enable high throughput of data, a read pipeline is implemented. This operates as a FIFO (first-in-first-out) of requests in which there is no final destination specified when the request is made. Requests are pre-issued into the FIFO and then removed for use by a subsequent instruction. This instruction may then issue a new request of its own, thus keeping the FIFO full at all times.

In practical terms, using the read pipeline is equivalent to splitting a load into two operations; issue and completion, allowing the Meta core's pipeline to execute instructions while a memory read is in progress. In the first phase, the address to be read from is issued to a read address port (RA*). In the second phase, the returned data may be read from the read data port (RD).

This increases performance for many applications because there is a buffer of requests waiting. The capacity to handle six reads is typical to support portable code, but the number read operations varies according to the specification of the Meta core and the software's requirements.

Memory read ports

Each thread has its own set of read ports, which consist of both output (e.g. read address issue) and input (e.g. read data fetch ports). Memory read pipelines are private to each thread and other threads cannot change the state contained in them.

To obtain pipelined read data from memory a thread issues a 32-bit read address then the data associated with that address is read back into the Meta core. All read ports support word-sized accesses to 8, 16, 32 or 64-bit word boundaries.

Input	Output
Read Data (RD).	Read Address (RA).
Reserved for future.	Read Address Prefetch (RAPF).
Reserved for future.	Read Address Byte Zero Extend (RABZ).
Reserved for future.	Read Address Word Zero Extend (RAWZ).
Reserved for future.	Read Address Dword Zero Extend (RADZ).
Reserved for future.	Read Address Byte Sign Extend (RABX).
Reserved for future.	Read Address Word Sign Extend (RAWX).
Reserved for future.	Read Address Dword Sign Extend (RADX).
Reserved for future.	Read Address MX Extend 8-bit (RAM8X).
Reserved for future.	Read Address MX Extend 16-bit (RAM16X).
Reserved for future.	Read Address MX Extend 8x32 (RAM8X32).

Under most conditions only the read data input port and read address output port need to be used.

The read address prefetch output port allows speculative and predictive pre-loading of the data cache. The read port's zero and sign extension variants allow byte, word or dword reads to be zero or sign extended before use. Specifying this with the read address issue makes it possible to apply zero or sign extension to instructions that would otherwise not support it. To use these extensions the read address is issued to one of the special read ports and the data fetched from the RD port is modified accordingly.

DSP memory read ports

A number of DSP instructions support an additional memory port feature, the MX flag, which indicates that the memory operation (load or store) is targeted towards 16-bit or 8-bit operations. See the Meta DSP TRM for details.

Memory load ports

Each thread has its own load port that can queue and complete a sequence of reads from memory into specified registers in the background while the processor continues to operate and execute other instructions. If any of the data values requested are required by a thread, then the scheduler will delay

its execution until the data required arrives into the related register. Load operation can deliver zero extended 8, 16, 32, or 64-bit word data from corresponding word boundaries into a register in any unit.

Memory write port

The single memory write-port is driven from the write buffer and shared between all threads. It supports writes of 8, 16, 32, or 64-bit words aligned to word boundaries. Write addresses are generated as 32-bits.

Coprocessor write ports

The Meta core supports up to eight coprocessor write ports.

A coprocessor write port is a uni-directional channel from the Meta core into a coprocessor hardware module. This channel may have an arbitrary width up to 64-bits. The instruction set includes encoding that is suited to operating with both full 64-bit ports and reduced width ports (e.g. 48-bit). Coprocessors may optionally support full flow control on these types of ports, with buffering commonly being added to flow controlled ports to improve throughput.

Coprocessor read ports

The Meta core supports up to eight coprocessor read ports, these ports usually have a relationship to the corresponding coprocessor write ports.

Coprocessor read ports are typically 64-bit or 32-bit wide uni-directional channels from a coprocessor into the Meta core. Flow control is supported by these ports (except in exceptional circumstances), as these ports are emptied only when demanded by the software running on one of the Meta core's threads.

2.4. Exceptions, Triggers and Kicks

During code execution a number of events can occur both externally and internally that can be observed and acted on in multiple ways by the Meta core. This section overviews the events that the Meta core can generate itself or observe whilst is operates and the techniques that can then be deployed to respond to specific sub-sets of these events. Conventional processors respond to events by running interrupt service routines and Meta cores support these standard methods as well as other advanced methods of event handling.

Terminology

The term *trigger* is used both to refer to general purpose interrupt signals fed into the core and specialised internal events that the core generates such as exceptions and kicks. The Meta core responds similarly via a number of common mechanisms to events of all types so the term *trigger* is used as an encompassing term for all events which are relevant to the context concerned rather than enumerating all the types of event that could be involved at each point in the text.

The terms *exception* and *interrupt* are used for events that alter the normal program flow to handle odd conditions or triggers that disturb the normal execution flow or actually prevent instructions completing normally. Interrupts are normally used to handle asynchronous events, while exceptions handle conditions detected by the processor itself in the course of executing instructions. However, exceptions and interrupts are often used inter-changeably, and the term *exception* is often used to refer both to exceptions and interrupts.

Some instructions are designed to always provoke specific exceptions, these instructions are commonly called *traps* and in the Meta architecture such operations are implemented by SWITCH instructions.

Both interrupts and exceptions can cause the thread to alter the program flow to execute an *interrupt handler* or *exception handler* to manage such events in a fresh execution context described as *interrupt level* processing. The original *background level* execution of the thread is usually unaffected by the presence of interrupt level processing events and lower priority interrupt execution is unaffected by any higher priority interrupt level processing events that interrupt it. Each interrupt processing level is hence a clean and independent execution context that can restore all state related to the previous execution level when it completes.

The system responds to an *exception* (i.e. exception or interrupt) event by automatically saving critical execution state onto a stack. The stack on which state is stored is referred to as the *interrupt stack*. The execution state saved onto the interrupt stack forms the *interrupt frame*. The process of saving the execution state is called the *entry sequence*. When the thread completes the interrupt level code, the process of restoring execution state is called the *exit sequence*. When an exception pre-empts an instruction stream, the thread automatically saves execution state into the interrupt frame, and execution branches to the corresponding exception handler. Exception priorities determine the order in which the thread handles exceptions. Exceptions of higher priority can pre-empt the instruction stream of a currently executing exception handler via nested interrupt processing. Exceptions of lower priority are pending exceptions which will be automatically handled after the current exception handler is completed.

2.4.1. Advanced Trigger Processing

Interrupt handling can have a large overhead involving a save of the current context, execution of the interrupt service routine, and a restore.

With advanced trigger processing, threads can poll or wait for events and respond immediately.

For multi-threaded versions of the Meta core, code previously invoked via interrupt handlers may be placed on a different thread. In this case, no context save is required, so there is a true one-cycle response without overheads. Simple code on one thread can respond to multiple events synchronously.

Trigger masks are used to select the trigger sources that the thread wishes to respond to and status registers indicate pending events that a thread has yet to acknowledge.

The action of a thread is to either block or poll on one or more trigger sources. Blocking effectively deschedules the thread until the block is released and no processor load is used on that thread until the blocking condition is met.

An example of a trigger source is the internal timer that stalls a thread's instruction stream until a timeout occurs. The Meta core supports advanced trigger processing of two distinct types of trigger source via the trigger unit - hardware triggers and kicks.

Hardware triggers

Hardware triggers are sourced from hardware outside of the Meta core (e.g. coprocessors and external peripherals). Each hardware trigger provides a simple event flag.

A set of registers in the Trigger Control Register region are used to control the routing of internally generated and externally supplied triggers.

Kicks

Kicks are caused by a software action such as writing to a memory mapped register called a kick counter such as T0KICK (see section 4.2). Kicks differ from hardware triggers in that a count of kicks received is accumulated with time and automatically decremented (by one) when the thread responds. This counter can be used to implement simple software or hardware request queues, using shared memory or a coprocessor interface FIFO as the storage area. All software inter-thread events must be communicated using kicks.

A separate kick counter is supported for background use by each thread. Background processing allows zero-overhead, voluntary processing of kicks.

Timer Trigger

Timer triggers are generated by the timer register inside Meta core. A TXTIMER register is provided for independent use within background level code on each thread generating timer trigger events. For each thread the trigger unit holds two background trigger handling registers - namely the status/clear and mask register. The trigger mask is used to select the trigger sources that the thread wishes to respond to (for each interesting trigger source a '1' must be set in the relevant mask register bit), these sources include both hardware and kick type events (1-bit per source). All threads can address their background trigger mask register as TXMASK and the status/clear register TXSTAT is then used to receive and acknowledge triggers. To support polling accesses read-only access via the TXPOLL registers return the same status information as TXSTAT, but via a non-blocking read.

The background trigger status/clear registers allows both a blocking and non-blocking method for detecting fired triggers. The more common use is the blocking form where the thread's instruction stream will block until any of the trigger sources specified in the trigger mask TXMASK has a change

of state (i.e. an event is sent from that trigger source to the Meta core). All threads can address their background trigger status/clear register as TXSTAT (although this can only support the blocking form for reads from the background trigger status register).

TXSTAT only contains '1' bits that are selected via similar '1' bits in TXMASK. Triggers on the bits currently ignored are still detected and recorded within hardware common to all the threads so that when further '1' bits are enabled in a thread's TXMASK register the TXSTAT value reported for that thread may immediately change.

The blocking instruction will typically be of the form of a unit-to-unit move with the trigger unit as source and one of the address or data units as destination. In this case, after the blocking instruction is passed the move's destination register will contain both kick and trigger status (kick status being in the top half word, and trigger status being in the low half word). A threads kick count will be automatically decremented by one at the point at which it is read provided that the relevant bit was set in the trigger mask, one or more kicks have been accumulated and the kick event is the highest priority trigger source visible at the time.

To clear background triggers that are no longer needed the TXSTAT register is written to with a bit pattern that reflects the acknowledged triggers (exactly the same form as per the trigger mask). Writing a '1' will clear the relevant trigger, while a '0' will leave the trigger state unchanged. Attempting to write to the kick status bit or accumulator bits will not change the kick count in any way (accumulated kicks can only be removed by a blocking trigger status read instruction when they are the highest priority event visible at the time).

To successfully acknowledge TXSTAT bits the corresponding TXMASK bits must be set. TXMASK defines which trigger bits the thread concerned is currently interested in and has influence over at background processing level.

To poll on background trigger state an additional pseudo-register TXPOLL is provided. Requesting a transfer from this register to another core register, for example in D0, D1, A0 or A1, results in a non-blocking read being performed. The state information returned for this request is in the same form as that retrieved from a blocking read. When polling the kick count is not decremented.

2.4.2. Trigger allocation

The bits within the trigger mask and status/clear registers are allocated/used as follows:

TXSTAT/TXSTATI, TXPOLL/TXPOLLI

Bits	Associated Trigger/State	Scope
31:24	Deferred Bus Error State (read only)	Independent
23	Deferred Bus Errors / linked test & set.	Independent
22	0	Reserved
21	Floating Point denormal trigger.	Independent
20	Floating Point invalid operation trigger.	Independent
19	Floating Point divide by zero trigger.	Independent
18	Floating Point overflow trigger.	Independent
17	Floating Point underflow trigger.	Independent
16	Floating Point inexact trigger.	Independent
15:4	Hardware trigger 15 to Hardware trigger 4	Global common
3	Deferred Exception trigger (not edge sensitive)	Independent
2*	Internal Background HALT trigger (* TXSTATI / TXPOLLI only)	Local common
1	Internal kick non-zero trigger (not edge sensitive)	Independent
0	Internal timer trigger	Independent

TXMASK/TXMASKI

Bits	Associated Trigger/State	Scope
31	This bit is always set to '0'.	Reserved
30:16	These bits are always set to '0'.	Reserved
15:4	Hardware trigger 15 to Hardware trigger 4	Global common
3	Deferred Exception trigger (not edge sensitive)	Independent
2*	Internal Background HALT trigger (* TXMASKI only)	Local common
1	Internal kick non-zero trigger (not edge sensitive)	Independent
0	Internal timer trigger	Independent

Global common triggers may only be enabled in one of the interrupt or background trigger masks of one of the threads in the system at any time.

Local common triggers may only be enabled in one of the interrupt or background trigger masks for each thread in the system at any time. The HALT trigger in this category can only be usefully enabled in the interrupt trigger mask; enabling this event for background trigger processing is pointless as the occurrence of this trigger signifies that the background processing level of the thread has been stopped. If a HALT condition arises during interrupt level processing then the HALT trigger state can only be usefully detected externally to the Meta core or by an independent thread.

Independent triggers may be enabled in all the trigger masks of all the threads in the system. These bits support trigger hardware local to both each thread and each processing level of a thread.

Interrupt Only fields only exist in the TXMASKI register and hence may only be set in the interrupt trigger masks of all the threads in the system. These bits support trigger related hardware state local to each thread which only effect interrupt level processing.

2.4.3. Trigger matrixing

To allow some fault conditions on one thread to be handled via an interrupt on another thread, a set of triggers may be matrixed to any thread.

These triggers are:

Background HALT	Only issued once pipeline and memory state has stabilised.
Interrupt HALT	Only issued once pipeline and memory state has stabilised.
Background or Interrupt HALT	Which was caused by a page fault or read only fault on code or data access. Only issued once pipeline and memory state has stabilised.
AMA trigger	This trigger indicates that certain conditions within the AMA rate/priority control mechanism have occurred (see AMA register 0 - TXAMAREG0).

Classing the above triggers as B, I, and P the Meta core outputs a set of triggers to be matrixed as shown below:

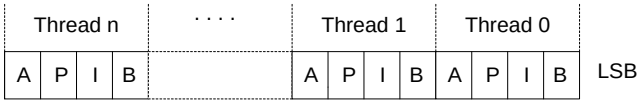

Figure 10 Trigger matrixing

Note: The thread trigger slot labelled A in the diagram is reserved.

These triggers can be vectored in a way similar to triggers completely external to the processor to effect any one of the twelve trigger lines 4 to 15 fed into the threads trigger mechanisms.

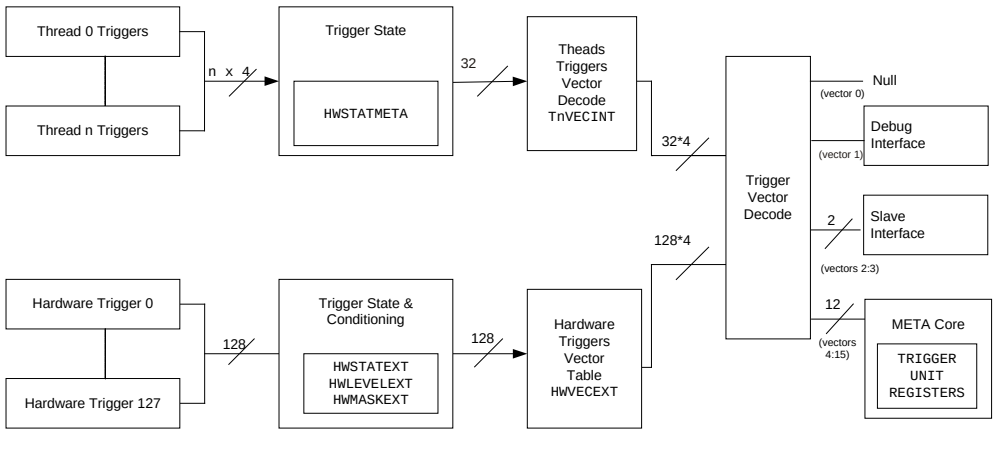

Figure 11 Trigger flow

2.4.4. Interrupt triggers

Meta interrupts include hardware triggers from outside the core, events triggered by the internal timer and KICK events accumulated for interrupt level handling by that thread.

Hardware triggers

Hardware triggers are sourced both from hardware outside of the Meta core itself (e.g. coprocessors) and internal sources (e.g. timer). Each hardware trigger provides a simple event flag.

A set of registers in the Trigger Control Register region are used to control the routing of internally generated and externally supplied triggers.

Kicks

Kicks are caused by a software action such as writing to a memory mapped register called a kick counter such as T0KICKI (see section 4.2). Kicks differ from hardware triggers in that a count of kicks received is accumulated with time and automatically decremented (by one) when the thread responds. This counter can be used to implement simple software or hardware request queues, using shared memory or a coprocessor interface FIFO as the storage area. All software inter-thread events must be communicated using kicks.

A separate kick counter is supported for interrupt use by each thread. Interrupt processing of kicks allows operating system or interrupt handling code on one thread to communicate transparently with similar interrupt handling software on any other thread.

For each thread to operate independently at interrupt level the trigger unit holds two registers - namely the status/clear and mask registers for interrupt triggers. The trigger masks select the trigger sources that the thread wishes to respond to at interrupt level (for each interesting trigger source a '1' must be set in the relevant mask register bit), these sources include both hardware and kick type events (1-bit per source). All threads can address their interrupt trigger mask register as TXMASKI. The interrupt status/clear register TXSTATI is used to receive and acknowledge interrupts. To support polling accesses a read-only TXPOLLI pseudo register is provided that return the current status information via a non-blocking read.

The interrupt trigger mask register specifies the trigger sources that can cause the normal execution of the thread concerned to be interrupted. The basic interrupt mechanism solely involves switching PC and PCX for the affected thread.

The interrupt trigger status/clear registers allow interrupt triggers to be detected and then selectively acknowledged in a way similar to background trigger states. If significant interrupt trigger states still exist when the exit sequence is invoked then the interrupt handling process is immediately re-triggered. All threads can address their interrupt trigger status/clear register as TXSTATI in the trigger unit. As with the background trigger registers it is possible to perform either a blocking or a polling operation upon the status/clear register. For blocking access, register TXSTATI may be read directly.

To poll on interrupt trigger state an additional pseudo-register TXPOLLI is provided. Requesting a transfer from this register to another processor register (e.g. in D0, D1, A0 or A1) results in a non-blocking read being performed. The state information returned for this request is in the same form as that retrieved from a blocking read.

2.4.5. Exceptions

Exceptions cause execution to halt and are then matrixed to allow the interrupt for one thread to be handled by another thread. This means that an exception may be handled by the thread on which it occurs, or on any other thread, depending on how the originating thread is set up.

Possible causes for an exception can be:

- Memory fault on data memory port, instruction memory port or hardware breakpoint.
- Unknown instruction or bad instruction pattern.
- Software traps (SWITCH instructions).
- Privilege violation.
- Instruction violations – instructions can become invalid in certain circumstance.
- Read pipeline overflow or underflow.
- Trigger blocking read that cannot be passed – reading from TXSTAT when TXMASK is zero.
- Deferred floating point exception.
- Memory bus error.

For writes to control (CT) registers from inside the core (register transfer or load from memory), if a CT register requires privilege to be written to, and the thread does not possess the requisite privilege, an exception is raised instead of the write being performed. In general, privilege controls for CT registers are either controlled by bits in TXPRIVEXT or have a static required/not required rule.

SWAPs between any of units CT/PC/TR/TT and similar CT/PC/TR/TT units also cause an exception as normal address and data unit registers must be used instead as intermediaries in any such specialist unit transfers.

2.4.6. Deferred triggers

The Meta core trigger and exception model has been augmented to include the concept of deferred exceptions. This applies to two particular areas – supporting memory bus errors (including test and set style functions) and floating point (e.g. for divide by zero). The possible reasons for a deferred trigger are as follows:

- Deferred bus error – this typically relates to a read or write to memory that generates an error state. A sub-set of bus errors are called bus warnings and the most common cause of this type of bus error state is linked test and set where both the success and failure states are 'expected' conditions during normal operation .
- Floating point inexact exception.
- Floating point underflow exception.
- Floating point overflow exception.
- Floating point divide by zero exception.
- Floating point invalid operation exception.
- Floating point denormal exception.

For more details of floating point deferred exception handling refer to the Meta FPU TRM

Direction into Interrupt or Background Level

Deferred triggers may be handled at background level or via interrupts. A single control (per reason) is provided that allocates whether an exception causes an interrupt or not, with the default setting for this being that deferred triggers are assigned to background level and will not cause an interrupt to be raised. The background/interrupt selection is controlled via the TXDEFR register.

Trigger Masking and Status

In the trigger unit two trigger controls exist in bit 3 of TXMASK/TXMASKI as shown in section 2.4.2.

This trigger mask bit is by default in the off (masked) state and may be set to enable the deferred triggers (as routed to interrupt or background level by TXDEFR) to then lead to the raising of an interrupt (TXMASKI) or background trigger (TXMASK). This trigger control works in the same way as the existing bits.

For interrupt level triggers the extra trigger actually feeds into a thread's HALT mechanisms. It therefore slots into the IRQ priority encoding scheme so that software can make simple choices about what to deal with in cases where multiple triggers have triggered at the same time (see description of IRQEnc field of register TXDIVTIME, section 4.1.12).

Timer/catch state control

To determine which of the deferred trigger sources have caused a particular deferred trigger event a new set of status data has been multiplexed with the kick data when reading TXSTAT/TXPOLL/TXSTATI/TXPOLLI. Kick data has precedence so that this deferred error status will only appear when deferred errors are the highest priority active trigger at a specific level, and bit 3 of TXMASK* is set (note that DEFR instructions are run as if bit 3 of TXMASK* is set). The deferred error bits that may be reported back via reads of TXSTAT*/TXPOLL* are bits 16 to 31 as shown in section 2.4.2.

Within the set of deferred exceptions deferred bus errors have precedence over deferred floating point exceptions. What this means is that when doing a DEFR, that DEFR will only read and clear out the deferred bus error if it has one and leave the FPU exceptions untouched. Only if there is no bus error will the FPU exceptions be read and cleared by the DEFR instruction.

Reads of TXSTAT*/TXPOLL* can return all the deferred error status for the appropriate interrupt or background processing level dependent on the ICtrl routing bits in TXDEFR bits 7 and 5 to 0. The value of bit 3 remains ‘1’ whilst any of the deferred error triggers remain unacknowledged.

Writes to TXSTAT* can be used to clear deferred error status as per the rest of the bits in TXSTAT. Each of the 7 reasons can be cleared individually by setting just that one bit; however, bit 3 will also need to be written at the same time. Writes to the upper bits (and bit 3) to clear deferred state are expected and allowed when TXMASK* bit 3 is zero. So, to clear the deferred bus error status in TXSTATI it is necessary to write a ‘1’ to bit 3 and a ‘1’ to bit 23 (at the same time) and this is allowed when TXMASKI is set to zero within an interrupt level exception handler. Note that writing just bit 3 will have no significant effect.

The DEFR instruction provides a way to read TXSTAT* and clear deferred error states in a single operation. Care must be taken in an exception system to act on ALL the bits which are collected and cleared via a single DEFR instruction. The DEFR instruction can also be used with TXPOLL*, however, in this case the operation is a non-blocking read only (no deferred error state is cleared for DEFR against TXPOLL*). The more manual/selective mechanism of masked writes to TXSTAT* is used where appropriate, so, for example, state could be read using a DEFR on TXPOLL* and then a write to TXSTAT* using that data can be used to clear states. As noted above for TXSTAT*/TXPOLL* reads, the DEFR instruction will only read the set of deferred errors that apply to the selected processing level (background or interrupt) and this is determined from the register being read from (e.g. TXSTAT or TXSTATI). Also, DEFR will only clear the deferred errors that apply to the selected processing level – it will only clear those triggers it read.

TXMASK* means both TXMASK and TXMASKI etc.

2.5. Hardware response to Interrupts and exceptions

By default each thread within the Meta core will stop itself executing in response to an exception (i.e. HALT) and by default ignore any outstanding interrupts. Alternatively each thread can be enabled to execute an arbitrary interrupt level code in response to suitable sub-sets of exceptions, interrupts or both.

2.5.1. Low-level interrupt handling

If internal handling of triggers is enabled the code specified via the PCX register is run in a critical execution state indicated by the ISTAT bit of the TXSTATUS register being set to 1. Further pending interrupts are ignored whilst in this state so the code concerned is written to transfer control to a more normal execution state with PCX restored and ISTAT set back to 0 as soon as possible forming a loop terminated for each event by the execution of the special purpose return-from-interrupt RTI

instruction. This then allows the execution of arbitrary exception handling code in a state that enables nested handling of any further higher priority events, the enabling of nested handling within this context is optional as it is often just as efficient to complete the handling of the current event before responding to the next.

2.5.2. HALTS versus interrupts

All exceptions provoke a specific trigger private to each thread which may lead to the execution of the low-level interrupt handling code or provoke a HALT sequence.

HALTs occur when an instruction provokes a problem and it is not possible for this issue to be handled by the thread itself. This is either because it occurs at interrupt processing level (ISTAT=1) or because the trigger generated by the exception is not setup as an interrupt to be handled internally by the thread concerned.

Most of the logic associated with handling interrupts or HALTs is common because they both need to deal with saving the state of the currently executing code on the thread concerned. Once this state is saved, the two sequences diverge to either HALT and signal hardware external to the thread or alternatively start execution of the critical PCX addressed handling code inside the thread.

2.5.3. HALT/Interrupt sequence

When a HALT condition occurs a series of actions may be carried out for the affected thread.

A table showing the possible state changes is shown below:

Current State (num)	New State	Cause	Changes
OFF (0)	RUN	Thread turned on	TXENABLE CT reg.
RUN (1)	CBRESTORE	RTI with catch state to replay.	-
RUN (1)	TIDYOFF	Thread turned off	-
RUN (1)	TIDYIRQ	Interrupt occurred.	TXSTAT CT reg.
RUN (1)	TIDYHALT	Exception occurred.	-
CBRESTORE (2)	RUN	-	TXSTAT CT reg (CBMarker).
TIDYOFF (3)	OFF	Catch state is tidy.	External trigger(s) issued.
TIDYHALT (4)	HALTED	Catch state is tidy; thread will handle its own exceptions.	Internal HALT trigger issued.
TIDYHALT (4)	OFF	Catch state is tidy; thread will not handle its own exceptions.	TXENABLE CT reg, External trigger(s) issued.
TIDYIRQ (5)	RUN	Catch state is tidy.	-
HALTED (6)	OFF	Thread turned off.	TXENABLE CT reg, external trigger(s) issued.
HALTED (6)	RUN	Interrupt occurred.	TXSTAT CT reg.

When a HALT or interrupt occurs a series of actions are performed for the affected thread to preserve its current state.

Each thread may have a long-term memory state, for example pipelined read, and a transient memory state such as loads or writes. Each thread has a catch register that holds this current state. When a HALT occurs all execution pipelines must become stable and catch states must be saved after all memory activity has ceased.

When execution pipelines are stable and the catch state saved, a HALT trigger is raised which may either cause an interrupt on the affected thread or be matrixed, causing an interrupt on another thread or the debugger or another external processor. The thread remains HALTed until such time as the thread is restarted by another thread or debugger etc.

Interrupts must also be managed. If an interrupt occurs when a thread's catch state holds a non-transient state it must be saved before switching the thread's context.

Once an interrupt has been serviced, background catch state will be replayed if the thread's catch saved marker (CBMarker) is set when the return from interrupt (RTI) instruction is executed. When a thread's interrupt is actually handled by another thread (via HALT trigger matrixing) the thread's state may be replayed by the action of re-enabling the thread (again when the catch state saved marker is set).

For details of the catch state registers see section 4.1.7 Catch state register 0 - TXCATCH0.

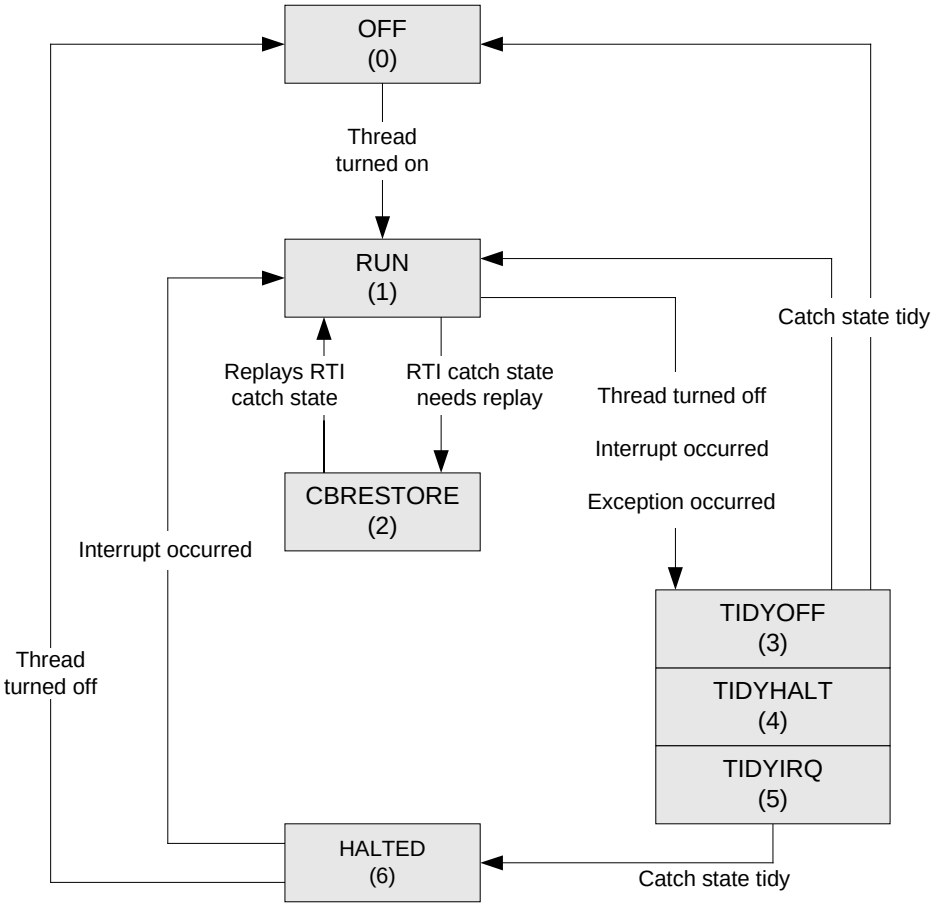

Figure 12 Catch state changes: OFF, RUN, CBRESTORE and HALTED

Note: TIDYOFF, TIDYHALT, and, TIDYIRQ are intermediate states only.

2.6. Instruction and data caches

The Meta core interfaces directly to code and data caches that support its full read and write operating bandwidth. Using this interface the core may issue both a data read/write operation and a code read operation in a single cycle, with the intention of moving data into and out of the corresponding caches as fast as possible in parallel.

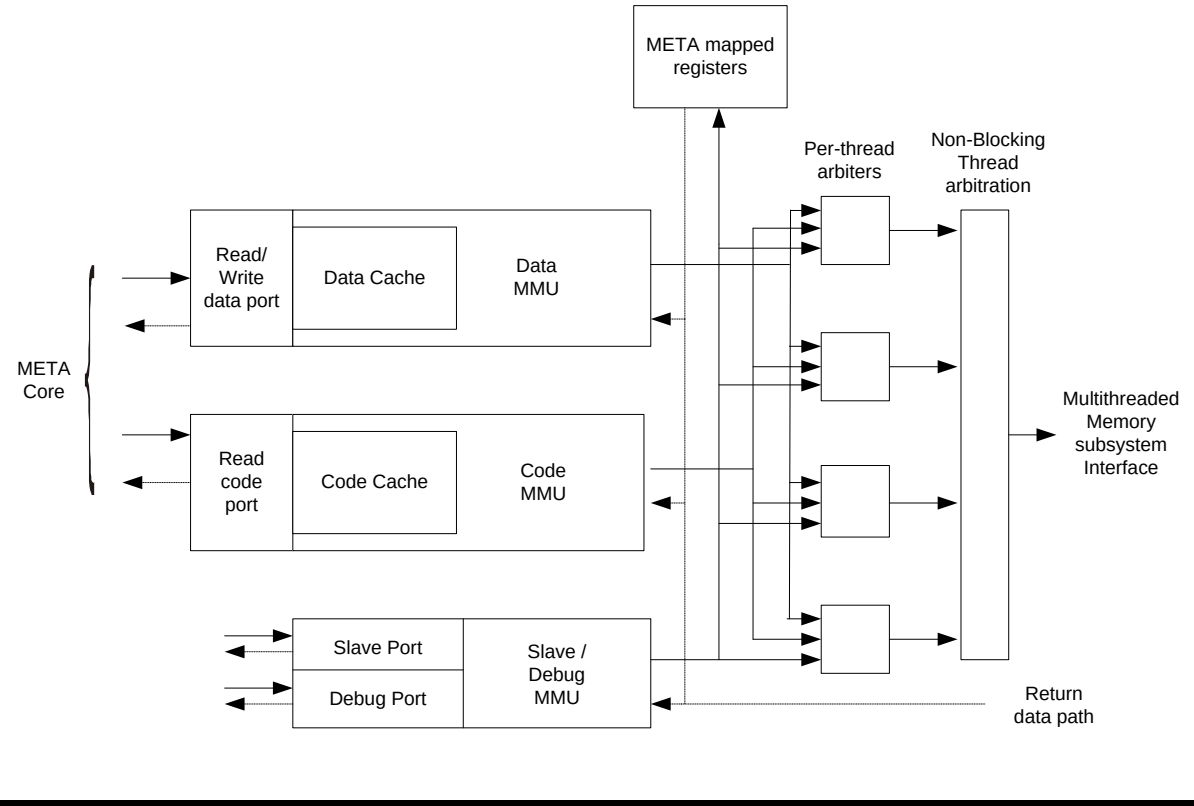

Figure 13 HTP Cache/MMU overview

Following a read request, information stored in MMU control registers and associated page tables indicates if the data cache or instruction cache should subsequently allocate space to retain the data returned.

The MMU responds to the cache indicating whether or not the address specified in a read or write request is valid. This allows hardware such as the data cache to operate in a way that is synchronised with each memory transaction on the MMU, allowing invalid writes to be detected before thread execution is allowed to continue. Various sorts of invalid transaction can be detected by the MMU, so additional data is provided by the MMU to describe why each transaction failed.

2.6.1. Cache organisation

The Meta HTP core has separate instruction and data caches that are virtually indexed, physically tagged (VIPT). Requests from the core have their linear address translated to a physical address before checking whether it is a cache hit or miss by comparing the physical address from the tag RAM with the translated address. The caches are both 4-way set associative. The cache line size is 64-bytes with a maximum of 512 lines in total.

The size of each cache is fixed at 4kbytes, 8kbytes, 16kbytes, 32kbytes or 64kbytes depending on the particular implementation of the core.

The caches may be partitioned at run-time into halves, quarters, eighths, sixteenths and allocated per thread. A thread has exclusive use of a local cache partition and all threads share a global cache partition.

The global data cache partition remains coherent for interleaved accesses by multiple threads without the use of cache invalidation.

The VIPT structure imposes a per-thread cache partition size in the physical cache of 16kbytes maximum when 4k pages are used or 32kbytes when 8k pages are used.

The caches are non-blocking on a thread basis, i.e. if one thread misses, requests by other threads will still be accepted.

The data cache operates with a write-through policy for all regions except for writes in the core-cached memory region. Locking of cache lines in this region is possible so that such cache lines operate effectively the same as core memories.

Data caches do not write-allocate. So if a cache write misses, a cache line is not allocated for the data written.

2.6.2. Cache manipulation

The Meta core provides features to allow the state of caches to be manipulated to improve performance or maintain coherency. If all memory update operations are performed to unique regions of the memory map then no cache manipulation is needed other than for performance enhancement.

In general cache manipulation operations need to be performed separately for each cache line sized region concerned; i.e. for each 64-byte sized and aligned region of the memory space.

Instruction cache prefetch

The instruction ICACHE allows for preloading the instruction cache in advance of when the code is required to be executed. Significant performance enhancements may be realised by making use of this instruction, in particular for systems with high memory latency. In general prefetches are only useful if code currently in cache can be executed at the same time as new code is being fetched.

Data cache prefetch

Prefetching of data into the data cache is supported to optimize data-read performance when necessary. Like the Icache prefetch, Dcache prefetches are only useful if data currently in cache can be processed while the prefetch completes. This creates a chain of related prefetches, so that new data is always being read into the cache while the previously prefetched data is processed. Prefetches can be preceded by an invalidate operation (see next) if the most recent data needs to be read for processing.

Each new prefetch interlocks with its predecessor so that only one prefetch is outstanding at any one moment for each thread.

This interlock is used as a barrier so that reads to an area that are being prefetched are delayed until the prefetch actually completes. This stops data being read more than once from external memory. Prefetches to the same address as the previous one can be used at the end of a chain of prefetches to interlock with the last real prefetch prior to input data processing.

Instruction cache invalidation

A number of mechanisms exist for instruction cache and TLB invalidation. For all types of instruction cache invalidations (i.e. physical/linear/TLB), to ensure an invalidation is complete, the invalidate must be followed by a CACHERL instruction to the instruction cache. By the time the CACHERL instruction completes, the invalidation is guaranteed to be complete.

Lines in the instruction cache can be invalidated by writing a zero to the memory location, LINSYSCFLUSH_ICACHE_LINE + offset. The offset must be a multiple of the cache line size (64 bytes).

The entire instruction cache can be invalidated by writing a one to the memory location SYSC_ICACHE_FLUSH.

The CACHEW instruction may be used to perform Icache line linear invalidation. (It also invalidates the Dcache and the instruction and data TLB)

Data cache invalidation

Shared modifiable regions that support data exchanges with external hardware or other threads require cache invalidation to be applied in order to access the most up to date information.

If an invalidate operation follows a prefetch operation for the same cache line, the result may be that the prefetch operation completes after the invalidate operation. Prefetched data must therefore be accessed before it is invalidated. Issuing another prefetch between the overlapping prefetch and invalidate operation also ensures that the latest data is accessed.

A number of mechanisms exist for data cache invalidation.

Lines in the data cache can be invalidated by writing a zero to memory location LINSYSCFLUSH_DCACHE_LINE + offset. The offset must be a multiple of the cache line size (64 bytes).

The entire data cache can be invalidated by writing a one to the memory location SYSC_DCACHE_FLUSH. This is only used for start of day (or post reset) cache initialisation and must be performed before the cache is enabled. See section 4.7.17.

The instruction DCACHE performs a linear data cache line invalidation.

The CACHEW instruction may be used to perform Dcache line linear invalidation. (It also invalidates the Icache and the instruction and data TLB)

Data cache coherence

Data cache prefetches allow a thread to perform reads and writes to external memory in parallel, this state is normally prevented by the cache logic to maintain a coherent state between the cache and real memory. Once data that has been prefetched has been read, it can be invalidated to remove any possible incoherent cache state.

If areas of memory that are prefetched lie in separate cache line sized areas of the memory map to those written to by the thread then no incoherent state will appear in the cache.

2.6.3. Cache WIN Modes

The caches have a number of caching modes, which may be controlled in three ways:

Cores which contain an MMU, may be controlled on a page by page basis by the page table entry.

When Enhanced bypass mode is being used (which is usually the case for cores configured with no MMU or during boot), the caching mode is controlled by the MMCU_LOCAL_EBCTRL and MMCU_GLOBAL_EBCTRL registers.

The caching mode may be independently controlled for each direct mapped region using the MMCU_DIRECTMAP0_ADDR - MMCU_DIRECTMAP3_ADDR_registers.

The available modes are listed below:

Name		Cache operation
00	WIN-0	No caching is performed. The Core will only make a single read request on the bus. In all other WIN Modes, reads are expanded into bursts.
01	WIN-1	Cache lines are allocated normally but can be immediately re-used for another location. It is possible to access large areas of memory of this type without losing the bulk of the more permanent cache content.
10	WIN-2	In this case only one 'way' of the cache resource is allocated for caching data read by each thread. The cache way used is the same as the thread number. For a four threaded core with 2 way cache, thread 2 and 3 would use cache way 0 and 1 respectively. If that particular cache line happens to be locked the operation continues uncached.
11	WIN-3	Full normal cache operation. If a miss requires the use of a set where all lines are locked, the operation will continue uncached.

2.7. MMU

The Memory Management Unit is responsible for translating linear addresses used by Meta threads into physical memory addresses recognised by the memory subsystem.

The threads on the Meta core each can access a different (local) version of the lower half of the address range 0x08000000-0x7FFFFFFF. There is a globally accessible address range at

0x88000000-0xFFFFFFFF. In Meta core implementations which utilise an MMU, these linear address ranges can be considered analogous to "virtual" memory often referred to on other architectures.

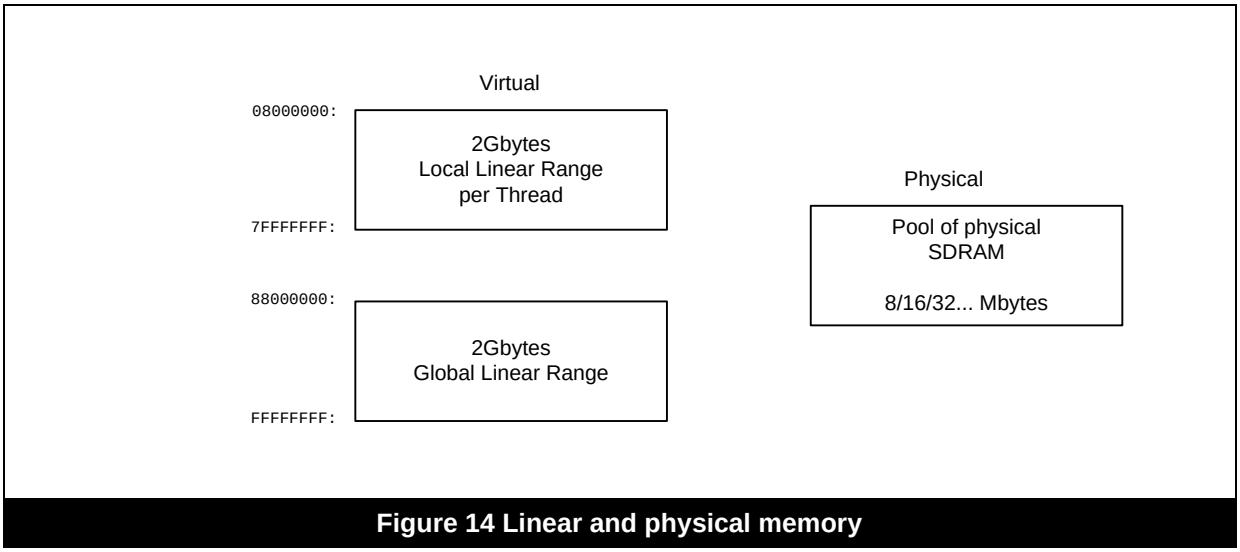

Figure 14 Linear and physical memory

The MMU page table region may be accessed using linear addresses in the region 0x05000000-0x05FFFFFF (LINSYSMTABLE_BASE - LINSYSMTABLE_LIMIT). These addresses are fixed in linear memory space.

Reads from the MMU page table region return a 32-bit page table entry that describes a valid MMU address, or error information in response to an invalid address to facilitate address checking and demand paging. An 'invalid' read is not treated as an error by the MMU so it is safe for any thread to read from any address within the MMU table region.

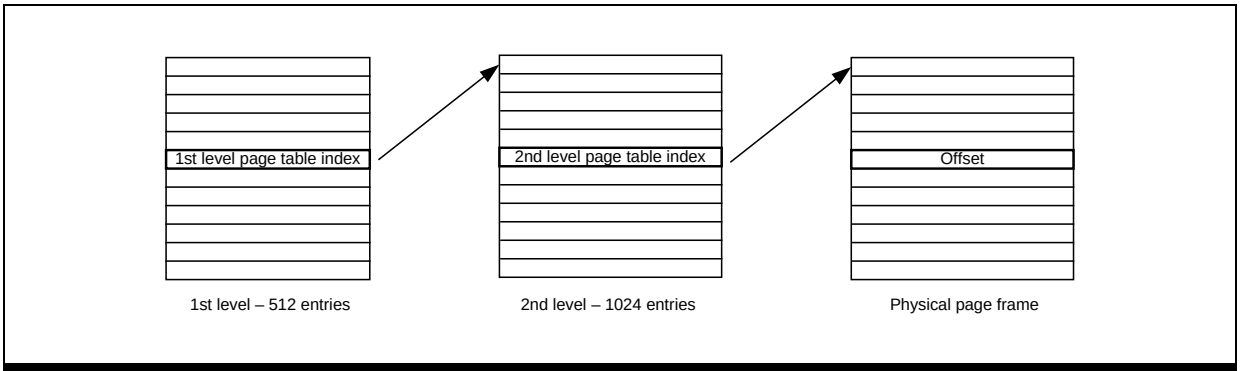

Figure 15 MMU page table layout

Linear addresses are decoded in the MMU as shown below:

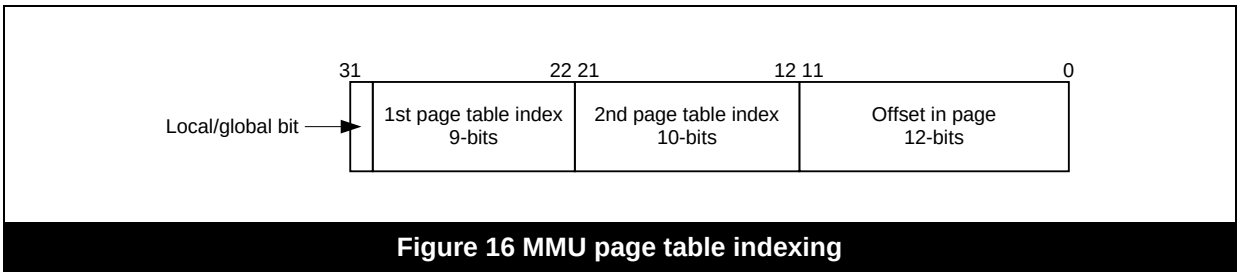

Figure 16 MMU page table indexing

The local/global bit specifies whether the 1st page table index bits should index the thread-local 1st-level page table or the global 1st-level page table.

0 = local page table

1 = global page table

2.7.1. Meta ATP MMU page table layout

HTP and MTP cores with an MMU support the page table structure used in the ATP core. Clearing bit-0 of MMCU_TABLE_PHYS_ADDR configures the MMU in ATP compatibility mode and the following structure applies.

The Meta ATP core has a single memory-mapped register named MMCU_TABLE_PHYS_ADDR, visible to all hardware threads. That register points to the base of a contiguous 12kbyte block of memory, 2kbytes for each 1st-level page table (512 entries x 4 bytes per-entry). Each table entry is associated with one 4kbyte page of physical memory and the table operates like a two-dimensional array.

Each 1st level entry contains the physical base address of the 2nd level entries.

Each 2nd level entry contains the physical address of the 2nd physical page frame.

1st-level page tables are 2kbytes in size with 512 entries. 2nd-level page tables are 4kbytes in size with 1024 entries. So, each 1st-level table entry address maps up to 4Mbytes of linear memory, and because physical page size in the Meta ATP core is 4kbytes, each first-level page table can map 2GB of virtual address space.

Each thread has a 2GB local region, and one 2GB global region, for a total of 4GB.

To assign a 4kbyte page of physical memory, first initialise the page table by entering the physical base table address in MMCU_TABLE_PHYS_ADDR.

Then fill in the 1st-level table entry with the physical base address of the 4kbyte page you want to assign. This activates 1024 2nd-level table entries, each of which is mapped to a 4kbyte page of physical memory.

The hierarchical nature of the table allows swapping of ‘address map data’ associated with a thread to be achieved via modification of only a few 32-bit entries.

2.7.2. Meta HTP MMU page table layout

The Meta HTP MMU is backward compatible with Meta ATP in that HTP can use an identical page table structure in physical memory. Setting a bit in MMCU_TABLE_PHYS MMU turns on an enhanced page table structure as described below which provides more flexibility between threads which allows for a more compressed page table in physical memory than was possible with ATP and page sizes other than 4k.

Note that the structure of the linear region 0x05XXXXXXX for accessing the page table is different between the ATP and HTP cores, but the structure of that region as described below must be used when the MMU is in ATP compatibility mode.

In the HTP page table structure, each hardware thread has four memory-mapped registers that are used to access that hardware thread's page tables: MMCU_TnLOCAL_TABLE_PHYS0, MMCU_TnLOCAL_TABLE_PHYS1, MMCU_TnGLOBAL_TABLE_PHYS0 and MMCU_TnGLOBAL_TABLE_PHYS1.These registers are the root of the page table structure.

The page table structure consists of three parts:

- A root table entry for each thread, for each of global and local region, which describes the base and the range of the linear address to be used and the physical base address of the first level entries which each describe a 4mbyte space This is held in the registers (above).
- 1st-level entry which contains the physical base address of the 2nd-level entries and the resolution of the 2nd-level entries (4kbyte to 4mbytes).
- 2nd-level entry which contains the physical base address of the linear address.

1st-level page tables are 2kbytes in size with 512 entries, so for each thread there are two 1st-level tables (local and global) mapping a 2mbyte region each. 2nd-level page table entries are 1024 lines and the resolution can be user configured from 4kbyte up to 4mbytes, but keeping the first level entries of 4mbytes resolution.

Note: If the MiniM bit is set, 2nd-level entries will be fewer than 1024.

The MMU's page table hierarchy can be accessed via the MMU Table region at addresses specified below. Each of the linear regions in the table below represents a 2Mbyte window to physical memory which may be arbitrarily placed but is 512kbyte aligned:

Linear Address	Name	Comment
0x05000000-0x051FFFFF	MMU Table Local Thread 0	A 2Mbyte memory region controlled by the value in MMCU_T0LOCAL_TABLE_PHYSx.
0x05200000-0x053FFFFF	MMU Table Local Thread 1	A 2Mbyte memory region controlled by the value in MMCU_T1LOCAL_TABLE_PHYSx.
0x05400000-0x055FFFFF	MMU Table Local Thread 2	A 2Mbyte memory region controlled by the value in MMCU_T2LOCAL_TABLE_PHYSx.
0x05600000-0x057FFFFF	MMU Table Local Thread 3	A 2Mbyte memory region controlled by the value in MMCU_T3LOCAL_TABLE_PHYSx.
0x05800000-0x059FFFFF	MMU Table Global Thread 0	A 2Mbyte memory region controlled by the value in MMCU_T0GLOBAL_TABLE_PHYSx.
0x05A00000-0x05BFFFFF	MMU Table Global Thread 1	A 2Mbyte memory region controlled by the value in MMCU_T1GLOBAL_TABLE_PHYSx.
0x05C00000-0x05DFFFFF	MMU Table Global Thread 2	A 2Mbyte memory region controlled by the value in MMCU_T2GLOBAL_TABLE_PHYSx.
0x05E00000-0x05FFFFFF	MMU Table Global Thread 3	A 2Mbyte memory region controlled by the value in MMCU_T3GLOBAL_TABLE_PHYSx.

Note: The physical address accessed via each region is a 512kbyte aligned address defined by bits 31:19 of the corresponding MMCU_Tn[LOCAL|GLOBAL]_TABLE_PHYS1 register.

The structure of this 0x05XXXXXX remains the same even when the MMU is configured in ATP compatible mode.

Manipulation of the MMU page table must always be accompanied by TLB flush for the affected threads whether or not it is accessed through the 0x05XXXXXX region.

The hardware checks whether the linear address issued is inside the lower and upper limit deduced from the range and base field of the MMCU_TnGLOBAL/LOCAL_TABLE_PHYS* registers. If it is out of the range, it reports a page fault to the core. The number of 1st-level entries at an offset to the physical base is determined by RANGE/4M.

When attempting to access a 1st-level table entry for an arbitrary linear address in either the local or global area:

- The linear base address and range values in the corresponding PHYS0 register must be consulted first as these define the range of linear addresses for which there is a valid mapping.
- Bits 18:2 of the MMCU_TnLOCAL_TABLE_PHYS1 register must be used as an offset to the start of the 1st-level MMU table within the linear range concerned.
- Data is then provided on the basis of one 32-bit entry for each 4MB region.

To avoid inter-thread dependencies, software running on each thread must only use the local or global region associated with that thread to access page table data. It is normal, but not prescribed by hardware, that all the global regions will refer to the same physical table in memory. If software wants to create or modify a global region mapping without affecting other threads, it must make a copy of the affected areas of 1st-level or 2nd-level table data during this process.

It is intended that when a 1st-level table data item is retrieved and decoded, that the physical address of the 2nd-level table data falls within the same 2mbyte window otherwise it will be inaccessible via the appropriate 0x05XXXXXX region. Tools that generate MMU page tables take this into account in their allocation policies. If an area that a thread needs to access falls outside both the local and global windows available then other means such as direct mapped areas must be used.

The regions may be protected and accessed in a cache-optimized mode as specified by bits 7:0 of the corresponding PHYS0 register as for other MMU table entries.

As direct read access of 2nd-level page table data can be achieved in an optimal fashion via use of the CACHERL instruction. Only software that wants to modify page table mapping changes or discover higher-level features of the page tables should use these regions.

Page table entry format

1st-level entries

The page table entry is 32-bits wide and the format is:

31:6	5	4:1	0
Physical Address Base of the 2nd-level entries. (Each block of 2nd level entries is 64-byte aligned)	MiniM 0 = off(default) 1 = on	2nd-level entry page size: 0 = 4k … 10 = 4M Others = Reserved	Valid

The number of 2nd-level entries at an offset to the physical base is determined by 4M/Page_Size.

In ATP MMU compatibility Mode bits 11:1 must be zero.

2nd-level entries

The page table entry is 32-bits wide and the format is:

31:12	11:8	7:6	5	4:0
Physical base address	Undefined - for software use	Cache control	Undefined - for software use	MMU control

Bits 31:12 of the entry specify the 4kbyte physical address base of the linear address.

The actual 32 bit physical address = (2nd-level entry & 0xFFFFF000) + (Linear_Addr & Mask)

where Mask = “00000FFF” if the entry is of 4kbyte resolution and Mask = “003FFFFF” if the entry is of 4M resolution etc.

MMU control bits

Bit		Name	Effects
0	MMCU_ENTRY_VAL_BIT	VALID	Always set for a valid MMU Table entry
1	MMCU_ENTRY_WR_BIT	WRITE	If set write operations to the region are enabled otherwise they are invalid.
2	MMCU_ENTRY_PRIV_BIT	PRIV	Set if only privileged requests are permitted within the region concerned.
3	MMCU_ENTRY_WRC_BIT	WR-COMBINE	Set if writes in this address region can be optimised as if targeted at external memory. Multiple small write transactions to adjacent of overlapping addresses can be optimised into larger transactions. If not set all writes must be performed in an unmodified sequence.
4	MMCU_ENTRY_EXM_BIT	EX-MODE	If not set exclusive reads and writes will only work amongst the threads of the Meta core. If set exclusive reads and writes will also work amongst the various devices in a system.

Cache control bits

These bits control the behaviour of the MMU and the related behaviour of the data or code cache.

Value		Name	Cache operation
00	MMCU_CWIN_UNCACHED	WIN-0	No caching is performed. The MMU only performs a read transaction in direct response to each read request made via the caches. In all other cases the MMU expands read operations into bursts.
01	MMCU_CWIN_BURST	WIN-1	Cache lines are allocated normally but can be immediately re-used for another location. It is possible to access large areas of memory of this type without losing the bulk of the more permanent cache content.
10	MMCU_CWIN_C1SET	WIN-2	In this case only one 'way' of the cache resource is allocated for caching data read by each thread. The cache way used is the same as the thread number. If that particular cache line happens to be locked the operation continues uncached.
11	MMCU_CWIN_CACHED	WIN-3	Full normal cache operation. If a miss requires the use of a set where all lines are locked, the operation will continue uncached.

2.7.3. TLB Invalidation

The TLB is effectively a cache of the MMU page table state within the Meta core. At times it becomes necessary to flush the TLB. This may be achieved by a write to the TLB flush region. See section 3.2.4 Cache / TLB invalidate for details.

2.7.4. Linear to physical address translation instruction CACHERL

The instruction, CACHERL returns the physical address page along with other flags for a given linear address.

If the linear address has no valid mapping (i.e. the 1st-level entry or the 2nd-level entry is invalid), the following data is returned:

Bit	Field	Comment
0	VALID	‘0’ – to indicate result is not valid
1	Undefined	Any value may be returned and should be ignored.
2	FIRST	‘1’ = First Level Entry Invalid, ‘0’ = First Level Entry Valid (hence second level must be invalid)
63:3	Undefined	Any value may be returned and should be ignored.

If the linear address has a valid mapping, the return data is as follows:

Bit	Name	Comment
0	VALID	Always set for a valid MMU Table entry – 1
1	WRITE	If set write operations to the region are enabled otherwise they are invalid.
2	PRIV	Set if only privileged requests are permitted within the region concerned.

Bit	Name	Comment
3	WR-COMBINE	Set if writes in this address region can be optimised as if targeted at external memory. Multiple small write transactions to adjacent overlapping addresses can be optimised into larger transactions. If not set all writes must be performed in an unmodified sequence.
4	SYS-COHERENT	Set if read or write operations must only be performed to the region once the WR-ATOMIC event has been issued to claim global access to the region concerned.
5	Undefined	Will reflect the contents of the 2nd level table entry (for S/W use)
7:6	Win Mode	Controls the cache win mode; e.g. 00 – uncached, 11 – fully cached.
11:8	Undefined	Will reflect the contents of the 2nd level table entry (for S/W use)
31:12	Phys Addr	The physical base address equivalent to the linear address specified to a 4kbyte resolution; physical address bits below this resolution come directly from the linear address.
32	Readable	1 = Readable, 0 = Write-Only
33	Reserved	Always returns 0.
34	Single-Use	1 = Entry to be deleted from the TLB after the first TLB entry hit, 0 = Retain Entry
35	Reserved	Always returns 0.
47:36	Page Size Mask	The entry page size(range from 4K=all1s, to 16M=all 0s)
63:48	Reserved	Always returns 0.

One use of this instruction is to deal with a memory fault/violation by issuing a CACHERL instruction to determine the type of fault/violation and its physical mapping. Then the new mapping data could be written to the physical location. A TLB linear invalidation will be needed if the previous mapping data had its VALID bit set to ensure the new mapping is fetched when the offending request is reissued; the TLB does not cache entries with the VALID bit not set.

2.7.5. MMU modes

The Meta MMU has three modes, bypass mode, enhanced bypass mode and active mode. The mode is controlled by the SYSC_CACHE_MMU_CONFIG register.

Bypass mode

On reset, the MMU starts in bypass mode. In this mode many of the features of the MMU are disabled and the MMU does not perform address translation. During system bootstrap in bypass mode the initial MMU Table must be created using physical memory accesses via the uncommitted physical address region. This allows the boot thread to initialise and arbitrarily define the content of memory subsequently used by all threads.

The memory map in section 3.1.3 shows the areas of the Meta core linear address space that are implemented while in bypass mode.

Enhanced-bypass mode

Enhanced bypass mode provides a simple method of enabling caches without requiring an MMU page table to be constructed. Enhanced bypass mode is enabled by setting the SYS_CACHE_MMU_CONFIG register to 0x6. The registers used to control enhanced bypass mode are MMCU_LOCAL_EBCTRL and MMCU_GLOBAL_EBCTRL.

The memory map in section 3.1.2 shows the areas of the Meta core linear address space that are implemented while in enhanced bypass mode.

Active mode

In active mode, the caches are enabled and the MMU performs address translation for code and data requests which have addresses in the Local and Global ranges. Active mode is enabled by setting the SYS_CACHE_MMU_CONFIG register to 0x7

2.8. Write combiner

The write combiner improves memory bus efficiency by amalgamating adjacent writes (which may be byte, word, dword or qword writes) into an appropriate burst format suitable for SDRAM and DDR type memory devices.

Note: Not all cores are fitted with write combiners.

2.9. Bus errors and Test/Set memory operations

The Meta HTP core memory interface is able to support features of IMGBUS 3.0 or other compatible buses to generate bus error information or perform linked test/set memory operations. These bus features are outlined below:

Bus Errors	A data stream of tokens that indicate the success or failure of each memory operation in an asynchronous but in-sequence manner. Normal memory operations that succeed may or may not return a confirmation according to the system concerned.
Linked Test/Set	A single read followed by a single write operation to the same critical address that implements an interruptible Test/Set S/W semaphore and returns an error if the write does not occur atomically with respect to the read; this operation may be used as an alternative to WR_ATOMIC to avoid real-time problems associated with stopping all memory operations.

The former bus error data stream is either permanently enabled or not as a function of the system infrastructure and has no run-time control. The latter linked read and write operation is identified by a side-band tag signal sent in parallel with the otherwise normal memory operation concerned.

To perform linked memory operations the processor provides a pair of instructions that can operate on 32-bit or 64-bit memory locations:

```
LNKGETD Reg,[BB.x]      or LNKGETL RegA,RegB,[BB.x]
…
LNKSETDcc [BB.x],Reg    or LNKSETLcc [BB.x],RegA,RegB
```

These instructions perform a linked-read and linked-write operation that can target external memory via the data cache. Both the read and write operations bypass the current data cache state neither hitting, flushing, or modifying the current cache state related to the address concerned. Use of these instructions to core data memory is not supported.

If the LNKSET operations condition code is not met then the operation will have no side-effects and no linked-write operation to the address concerned will occur. This NOOP behaviour will also occur if the thread concerned has been interrupted;

If issued the LNKSET operation provokes delivery of a single bus response token indicating success/failure of the overall sequence, if the sequence does fail then the write operation generated must not be committed to memory.

The following instruction is used to collect the deferred error data expected for this thread and sets the destination register appropriately:

```
DEFR Reg,TXSTAT         or DEFR Reg,TXPOLL
```

Reading from TXSTAT or TXPOLL controls whether the operation waits for all outstanding LNKGET/LNKSET response data to be received before providing a result. The TXPOLL case hence

checks transactions already completed without waiting for those still in flight whereas the TXSTAT case provides a complete result.

Note: that if no deferred error data is expected for the thread then the TXSTAT case operates in the same way as the TXPOLL case. It is hence safe for this unconditional instruction to follow a conditional LNKSET operation. If the LNKSET operation did not issue a linked memory write and trigger a deferred response then DEFR will not wait as no deferred response is outstanding.

Note: Bus error state for LNKSET success is a non-zero warning-level bus error code; whereas zero will be seen as the bus status if LNKSET was not issued. Once a significant pass/fail LNKSET bus error state is recovered with DEFR then the default zero state will be restored to underpin future LNKGET and LNKSET attempts.

Note: LNKSETs that fail the condition test (i.e. do not issue outside the core) return bus error state of 4.

Access to TXSTAT and TXPOLL by the DEFR instructions is allowed from a background execution environment when access to other interrupt-related logic is prohibited by privilege protection. Incorrect use of these instructions could permit an unprivileged program to delay its own execution, but that is not seen as a threat to normal system behaviour.

Only one item of deferred bus error data item is retained by the system for each thread and any LNKGET/LNKSET data outstanding at the time of an interrupt will first be waited for and then discarded. Interrupts can be prevented by the S/W if required. HALTing a thread similarly waits for any outstanding deferred state to be resolved before the HALT proceeds to completion.

The deferred bus error system will only report the last significant item, if transactions succeed after a failure then the previous failure data will be retained until cleared.

To perform an interruptible 32-bit ‘test and set’ operation using the above features the code would be:

```
;  D1Ar1 is an address to perform Test/Set logic on using the mask in D0Ar2.

TestAndSet:    LNKGETD D0Re0,[D1Ar1]          ; Perform a linked read
               TST     D0Re0,D0Ar2            ; Wait for the result and check the mask
               ORZ     D0Re0,D0Re0,D0Ar2      ; Set the mask if currently clear
               LNKSETDZ        [D1Ar1],D0Re0  ; Perform a linked write if previously clear
               DEFR    D1Re0,TXSTAT           ; Wait for and get deferred error response
               LSRZ D1Re0, D1Re0, #24         ; Get bus error summary
               ANDZ D1Re0, D1Re0, #0x3f
               CMPZ D1Re0, #2                 ; 0x2 indicates LNKSET success
               BNZ             TestAndSet     ; Go back if Z flag is lost at any point
               ;
; Note local/global cache state must be flushed/updated before accessing related
; data items after the critical access required is won.
;
```

To support linked test and set the LNKGETD instruction combines the issue of a load from memory with the setting of some thread specific state that marks the progress of the linked test and set. This state will be cleared when a LNKSET instruction is run or if the thread is interrupted or otherwise changes state (e.g. is turned off by setting TXENABLE to 0).

The LNKSETD instruction combines the issue of a conditional write to memory with the testing of the thread specific state that marks the progress of the linked test and set. If the state associated with the linked test and set has been cleared, this conditional write will effectively fail the condition test (i.e. not happen).The DEFR instruction waits for any known pending operations (floating point or linked test and set) to complete before pulling out the current status of all the deferred errors. This instruction is nearly equivalent to a read of TXPOLL (i.e. background level only) for just the deferred error parts, however any read of significant state implicitly clears that state from the deferred error system so that acknowledgement of such state via a write to TXSTAT is unnecessary. Collection of multiple deferred errors (LNKSET and floating point) can be achieved via a single use of DEFR and this can also be used to purge outstanding errors from previous floating point or uncollected LNKSET operations, only errors selected for delivery to background code will be seen and acknowledged via a DEFR on TXSTAT/TXPOLL.

3. Memory map

3.1. Overview

The memory map of the Meta core is presented in a linear address space, which, when an MMU is present, can be flexibly mapped onto a corresponding physical address space. All system and peripheral registers in the Meta core have memory-mapped access. The memory map varies depending on the mode in which the MMU is operating.

3.1.1. MMU active mode

In this mode, the MMU is enabled and offers its full feature set including address translation, cache control and memory protection. For cores which include an MMU, this is the main mode of operation. For some Meta cores that do not include an MMU, this mode is not available.

Linear Address Space			Physical Address Space (System Bus)	
00000000-001FFFFF		Invalid Region 0	00000000-01FFFFFF	Reserved
00200000-07FFFFFF		System Region		
	00200000-01FFFFFF	Reserved (Invalid)		
	02000000-02FFFFFF	Custom Area	02000000-02FFFFFF	Custom area
	03000000-03FFFFFF	Expansion Area	03000000-07FFFFFF	Reserved
	04000000-043FFFFF	System Event		
	04400000-046FFFFF	Physical Cache Flush		
	04700000-047FFFFF	Physical TLB Flush		
	04800000-04FFFFFF	Core Register Area		
	05000000-05FFFFFF	MMU Table Region		
	06000000-07FFFFFF †	Direct Mapped (0-3)		
08000000-7FFFFFFF †		Local Range	08000000-FFFFFFFF	Uncommitted
80000000-81FFFFFF		Core Code Memory		
82000000-83FFFFFF		Core Data Memory		
84000000-87FFFFFF †		Reserved		
88000000-FFFDFFFF †		Global Range		
FFFE0000-FFFFFFFF		Invalid Region 1		

† *These linear address regions may be mapped into the uncommitted physical address range can be cached in instruction cache or data cache under MMU control*

3.1.2. MMU enhanced bypass mode

In this mode, the MMU does not perform linear to physical address translation or the cache control features of the page table, but a number of regions in the linear address space are active and the caches are available for use. For cores which include an MMU, this mode provides a method of faster boot whereby the caches may be enabled without first configuring an MMU page table in memory. For cores which do not include an MMU, this is the main mode of operation.

Linear Address Space			Physical Address Space (System Bus)	
00000000-001FFFFF		Invalid Region 0	00000000-01FFFFFF	Reserved
00200000-07FFFFFF		System Region		
	00200000-01FFFFFF	Reserved (Invalid)		
	02000000-02FFFFFF	Custom Area	02000000-02FFFFFF	Custom area
	03000000-03FFFFFF	Expansion Area	03000000-07FFFFFF	Reserved
	04000000-043FFFFF	System Event		
	04400000-046FFFFF	Physical Cache Flush		
	04700000-047FFFFF	Physical TLB Flush *		
	04800000-04FFFFFF	Core Register Area		
	05000000-05FFFFFF	Reserved (Invalid)		
	06000000-07FFFFFF	Direct Mapped (0-3)		
08000000-7FFFFFFF		Local Range **	08000000-7FFFFFFF	Local Uncommitted
80000000-81FFFFFF		Core Code Memory	80000000-87FFFFFF	Reserved
82000000-83FFFFFF		Core Data Memory		
84000000-87FFFFFF		Reserved (Invalid)		
88000000-FFFDFFFF		Global Range ***	88000000-FFFDFFFF	Global Uncommitted
FFFE0000-FFFFFFFF		Invalid Region 1	FFFE0000-FFFFFFFF	Reserved

* If no MMU is present, this region is invalid.

** Controlled by MMCU_LOCAL_EBCTRL register

*** Controlled by MMCU_GLOBAL_EBCTRL register

Requests to invalid regions cause a general exception.

Enhanced bypass mode operates the same for cores with or without an MMU except in that cores with an MMU can perform privilege checks on the various regions.

3.1.3. MMU bypass mode

This is the default mode in which all Meta cores start up following reset. In this mode, the main features of the MMU and caches are disabled.

Linear Address Space			Physical Address Space (System Bus)	
00000000-001FFFFF		Reserved	00000000-01FFFFFF	Reserved
00200000-07FFFFFF		System Region		
	00200000-01FFFFFF	Reserved		
	02000000-02FFFFFF	Custom Area	02000000-02FFFFFF	Custom area
	03000000-03FFFFFF	Expansion Area	03000000-07FFFFFF	Reserved
	04000000-043FFFFF	System Event		
	04400000-046FFFFF	Reserved		
	04700000-047FFFFF	Physical TLB Flush *		
	04800000-04FFFFFF	Core Register Area		
	05000000-07FFFFFF	Reserved		
08000000-7FFFFFFF		Local Range**	08000000-7FFFFFFF	Local Uncommitted
80000000-81FFFFFF		Core Code Memory	80000000-87FFFFFF	Reserved
82000000-83FFFFFF		Core Data Memory		
84000000-87FFFFFF		Reserved		
88000000-FFFDFFFF		Global Range**	88000000-FFFDFFFF	Global Uncommitted
FFFE0000-FFFFFFFF		Reserved	FFFE0000-FFFFFFFF	Reserved

* If no MMU is present, this region is invalid.

** For cores with no cache these regions have the same behaviour.

In bypass mode, memory requests to the reserved regions do not necessarily cause an exception.

3.1.4. Invalid region 1 and 0

00000000-001FFFFF and FFFE0000-FFFFFFFF

These regions protect the system to some degree from invalid addresses generated by incorrect software. These addresses are detected as invalid addresses.

3.1.5. System region

00200000-07FFFFFF

The System Region is used to control and address all internal states of Meta core. This region is described in more detail in

Note: In MMU bypass mode, some areas in this region are not active.

3.1.6. Local range

08000000-7FFFFFFF

Addresses in this range are mapped via the MMU using data local to each thread. Locations in this region can be cached separately for each thread and are not directly accessible by other threads. In MMU active mode, this region is controlled by the MMU page table. In enhanced bypass mode the region is controlled by the MMCU_LOCAL_EBCTRL register. In bypass mode, requests are passed straight through to the system memory bus port.

3.1.7. Core code memory region

80000000-81FFFFFF

Both Code and Data requests to this region are routed to the closely coupled core code memory port of the Meta core. Note Data requests to this region are not closely coupled. Hence it is recommended to only use this region for code.

3.1.8. Core data memory region

82000000-83FFFFFF

Data requests to this region are routed to the closely coupled core data memory port of the Meta core. Code requests to this region will cause a general exception.

3.1.9. Global range

88000000-FFFDFFFF

Addresses in this range are mapped via the MMU using data global to all threads. Locations in this region may be cached in a global part of the cache and used by all threads. In MMU active mode, this region is controlled by the MMU page table. In enhanced bypass mode the region is controlled by the MMCU_GLOBAL_EBCTRL register. In bypass mode, requests are passed straight through to the system memory bus port.

3.2. System region

3.2.1. Custom area

02000000-02FFFFFF

LINSYSCUSTOM_BASE - LINSYSCUSTOM_LIMIT

This region can be used as the interface for an SoC register or memory bus outside the Meta core attached to the core's physical memory interface. Requests made through this memory region are not cacheable or write-combinable.

3.2.2. Expansion area

03000000-03FFFFFF

LINSYSEXPAND_BASE - LINSYSEXPAND_LIMIT

In previous versions of the Meta core, this region was used to control hardware at the periphery of the core. It is anticipated that the expansion area region will be used for a different purpose in future. Software should no longer access any registers through this region and should instead access them through the core registers area (0x04800000 - 0x04FFFFFF) where there are now duplicates.

3.2.3. System event

04000000-043FFFFF

LINSYSEVENT_BASE - LINSYSEVENT_LIMIT

These addresses are used to implement system control events using the following encoding:

Bits	31-22	21-12	11-6	5-0
	0000 0100 00	00 0000 0000	Command	Xxxxxx

The operation command codes can be used via simple 32-bit write operations:

Event	Command	Function
WR-ATOMIC Address: LINSYSEVENT_WR_ATOMIC_UNLOCK LINSYSEVENT_WR_ATOMIC_LOCK	0-Unlock 1-Lock	Implements an interthread fence for use within a LOCK2-held state that gives the thread control of the memory sub-system. This only provides exclusivity from other threads within the core, not other cores in the system. This event gets used internally and then automatically translated to 'Fence Port Requests' (system even 33) to ensure appropriate ordering is maintained outside the core.

Event	Command	Function
WR-CACHE Address: LINSYSEVENT_WR_CACHE_DISABLE LINSYSEVENT_WR_CACHE_ENABLE	2-Disable 3-Enable	Allows cache line fill behaviour to be disabled and enabled for the thread concerned via the MMU. After reset, cache line fills are enabled by default as soon as the cache is enabled. Disabling line fills later causes the MMU and hence the cache to operate as if every cache read or write miss is from an un-cached (WIN0) region. This prevents any changes to the current cache line allocation. Handling of hits for the addresses already in the cache is unaffected. No disruption of previously issued transactions still being processed by the cache is required. This system event gets used internally and then filtered (no corresponding external event).
Flush	4	Flush write-combiners lines owned by the thread issuing the system event. This system event gets used internally and then converted to 'Flush Writes' (system event 32) event for external use. The intention of this system event is to ensure that requests find their way to their destinations in a timely manner in particular for the case where an external buffer would otherwise store write requests indefinitely. There are no guarantees as to the ordering of the completion of the flush with respect to traffic from other requesters.
Fence Writes	5	This system event gets used internally and then converted to 'Fence System Requests' (system event 34) event for external use. The intention of this system event is to provide some control on the ordering of writes from the Meta core to various slave devices in a system. A typical usage example is software (such as a Linux driver) running on a Meta thread performing a sequence of writes to memory would issue this fence command before issuing a write to, say, a 'start' register in a DMA controller that subsequently attempts to read the data previously written by the Meta core.
Reserved	6-31	Reserved for future 'internal' events.
Flush Writes	32 (See note 1)	External write combiners, buffers, or caches containing write requests for the thread that issues this system event will be flushed.

Event	Command	Function
Fence Port Requests	33 (See note 1)	Fence port requests – If this system event is issued, all previously issued writes across all of the Meta threads are serviced before any further requests from Meta after the event are allowed to continue. This system event is used by the global cache functionality within Meta to ensure coherency. It is also used by the atomic lock/unlock functions to ensure appropriate ordering between Meta threads outside of the core.
Fence System Requests	34 (See note 1)	Fence system requests - If this system event is issued, all previously issued writes by the thread issuing the fence must be serviced before any further requests from that thread are allowed to proceed.
Reserved	35-47 (See note 1)	For future use
Uncommitted	48-63 (See note 2)	These commands may be allocated externally in a system specific way to gain access to any specific System Bus features defined.

Notes:

1 32-47 are system events that directly translate to external hardware system events on the bus. These should not be used by software running on the core as they will generally not have the desired effect within the core. When exposed on the bus only bus address bits 10:6 are used for decoding system events. Bit 11 should be treated as don't care.

2 48-63 are system events that directly translate to external hardware system events on the bus. These may be used for customer specific purposes or bus translation wrappers. When exposed on the bus only bus address bits 10:6 are used for decoding system events. Bit 11 should be treated as don't care.

3 Reads in this region will always fail with a general protection error.

3.2.4. Cache / TLB invalidate

04400000-047FFFFF

LINSYSCFLUSH_BASE - LINSYSCFLUSH_LIMIT

Writes to this address region can cause the direct invalidation of physical cache lines held in Meta core caches. Writes to this address region will be consumed by the MMU and not passed on further to the memory interface sub-system.

Single 32-bit writes with zero data content are used to cause invalidation.

Any read transactions performed in this region will always fail reporting a general protection error.

Cache Invalidate

The following sub-regions in the physical Cache Flush area are allocated:

Base Addr	Description
04400000 LINSYSCFLUSH_DCACHE_LINE	Meta core data cache; minimum 64-byte stride.

Base Addr	Description
04500000 LINSYSCFLUSH_ICACHE_LINE	Meta core code cache; minimum 64-byte stride.
04600000-046FFFFF	Reserved

Writes to the Cache Flush area must generally be performed at a stride corresponding to the size of the physical cache lines of the cache concerned. . The cache lines are addressed as lines 0..n-1 from set 0, then lines n.. 2*n-1 from set 1, then lines 2*n.. 3*n-1 from set 2, then lines 3*n.. 4*n-1 from set 3. Example:

To invalidate a Meta core data cache of size 8 KB and with a cache line size of 64 bytes:

```
offset = 0
while (offset < cache size)
{
flush address = LINSYSCFLUSH_DCACHE_LINE + offset
write 0 to flush address
offset += 64 cache line size
}
```

TLB Invalidate

The TLB is partitioned and hard allocated per thread and thread-specific invalidation may be performed by using the following addresses:

Name	Base Addr	Description
SYSC_TLBFLUSH	0x04700000	All threads TLB invalidate
SYSC_T0TLBFLUSH	0x04700020	Thread 0 TLB invalidate
SYSC_T1TLBFLUSH	0x04700028	Thread 1 TLB invalidate
SYSC_T2TLBFLUSH	0x04700030	Thread 2 TLB invalidate
SYSC_T3TLBFLUSH	0x04700038	Thread 3 TLB invalidate

Both 1st-level and 2nd-level TLB entries are flushed in each case.

3.2.5. Core register region

The registers used to control the Meta core are located in this region. Full details are in section 0.

3.2.6. MMU table region

The MMU page table may be accessed through this region.

3.2.7. Direct mapped

This address range is used for simple allocation to memory-mapped hardware.

All memory-mapped hardware may be accessed at arbitrary linear addresses as described to the MMU. However, the MMU also provides direct mapped access to critical physical address regions via these addresses, so that physical devices can be accessed in a way independent of the MMU set-up.

This region is also supported by cores that do not include an MMU.

Addresses in this region are translated using a simple base + offset scheme. The region is divided in to four 8 Mbyte sub-regions. Each sub-region maps directly to an area with a physical base address supplied from the appropriate MMCU_DIRECTMAPn_ADDR register.

The lower order bits of the address are added to this base to form the physical address:

Linear address	Physical address
06000000-067FFFFF	DirectMap0[31:23] + lin_addr[22:0]
06800000-06FFFFFF	DirectMap1[31:23] + lin_addr[22:0]
07000000-077FFFFF	DirectMap2[31:23] + lin_addr[22:0]
07800000-07FFFFFF	DirectMap3[31:23] + lin_addr[22:0]

When the direct map is configured as cacheable, the local partition of the cache is used for all accesses through this region.

4. Core registers

4.1. Control unit internal registers

These are 32 control registers which reside in the control unit's register file for direct access. In assembler these registers can be referred as CT.0 to CT.31 or their aliases such as TXSTATUS and TXENABLE. All control unit registers of all threads can be globally accessed via the memory-mapped region allocated to these registers.

This table is a summary of the registers that the control unit contains for each thread. Addresses given in this section are for thread 0. This block of registers is repeated at intervals of 0x1000 for each thread.

Register	Write Privilege	Name	Default (reset) value
CT.0 - TXENABLE	Variable	Thread ID/Enable	Thread 0 = 0xXXXXXX01 Threads 1-3 = 0xXXXXXX02
CT.1 - TXMODE			
CT.2 - TXSTATUS	Variable	Thread Status bits	Thread 0 = 0x00020000 Threads 1-3 = 0x00000000
CT.3 - TXRPT	None	Repeat Count	0x00000000
CT.4 - TXTIMER	None	Background Timer	0x00000000
CT.5 - CT.12- Reserved			
CT.13 - TXTIMERI	Variable	Interrupt Timer	0x00000000
CT.14 - Reserved			
CT.15 - Reserved			
CT.16-19 - TXCATCH0	Privileged write	Catch state registers.	0x00000000
CT.20 - TXDEFR	Variable	Deferred interrupt control.	0x00000000
CT.21 - Resevered			
CT.22 - Reserved			
CT.23 - Reserved			
CT.24 - Reserved			
CT.25 - Reserved			
CT.26 - Reserved			
CT.27 - Reserved			
CT.28 - TXDIVTIME	Variable	Timer Divider	0x00000001
CT.29 - TXPRIVEXT	Privileged write	Privilege Extensions/Step	0x00000000

Register	Write Privilege	Name	Default (reset) value
CT.30 - TXTACTCYC	Variable	Thread Active Cycles	0x00000000
CT.31 - TXIDLECYC	Variable	Core Idle Cycles	0x00000000

4.1.1. Thread enable - TXENABLE

Address: 04800000
Reset Value: 0xXXXXXX01 (thread 0), 0xXXXXXX02 (other threads)
Write Privilege: Variable

Bit	Symbol	Description
31:24 [READ ONLY]	MetaMajRev	Meta core major revision number.
23:16 [READ ONLY]	MetaMinRev	Meta core minor revision number.
15:12 [READ ONLY]	MetaTCaps	Meta core thread capabilities. See thread capabilities table below for values.
11:10 [READ ONLY]	Reserved	These bits are always '0'.
9:8 [READ ONLY]	MetaThreadID	Meta core thread number.
7:4 [READ ONLY]	MetaStepRev	Meta core step revision number.
3 [READ ONLY]	Reserved	This bit is always '0'.
2 [READ ONLY]	TStopped	This bit is set to 1 if the thread was stopped by clearing the threads ThreadEnable bit. If the thread is running or stopped due to a HALT this bit will be '0'
1 [READ ONLY]	TOff	This bit will be 1 when a thread's master state machine is in the off state. Seeing a 0 when a thread is known to have stopped indicates that the thread is still dumping state.
0	ThreadEnable	0 Thread is disabled. 1 Thread is enabled. In cases when buffer state has been saved by a thread it will caused to be reloaded when the thread is restarted (enable switched from 0 to 1). Conversely, if a thread is stopped (enable switched from 1 to 0) when there is long-term state held in the thread's catch (e.g. pipelined reads) the catch state will be written to memory and the “catch state” marker will be set.

Thread capabilities field of TXENABLE expanded

CAPS	DSP/FPU	DU Registers	AU Regs	FPU Regs
Threads with ‘Standard DSP’ features. Actual DSP configuration determined from DSP_TYPE field in CORE_ID register along with the BASELINE_DSP field in CORE_CONFIG2 register.				
0x8	DSP	9+9 (*)	4+4	
0xA	DSP+LFPU	9+9 (*)	4+4	16
0xB	DSP+LFPU	9+9 (*)	4+4	8
Threads with No DSP				

CAPS	DSP/FPU	DU Registers	AU Regs	FPU Regs
0xC	[FPU]	8+8	4+4	[16]
0xD	FPU	8+8	4+4	[8]
0xE	LFPU	8+8	4+4	16
0xF	LFPU	8+8	4+4	8
Threads with Extended DSP				
0x0	EDSP [+FPU]	16+16	8+8	[16]
0x1	EDSP+FPU	16+16	8+8	8

(*) For DSP_TYPEs specified that do not included the extended feature set (e.g. DSP_TYPE = 1 indicated in CORE_ID register) the DSP register set is 8+8 if BASELINE_DSP is 0 in CORE_CONFIG2 register or 9+9 if BASELINE_DSP is 1 in CORE_CONFIG2 register.

4.1.2. Thread mode bits - TXMODE

Address: 04800008
Reset Value: 0x00000000
Write Privilege: none

Bit	Symbol	Description
31:24	Reserved	These bits are DSP specific.
23	RCoProEn	This bit must be set before related transfer operations are performed (see XFR instruction). When set the data source for the XFR instruction is a coprocessor port.
22:0	Reserved	These bits are DSP specific.

Notes:

1 All thread mode option bits are ignored during interrupt state (i.e. after entering an interrupt service routine and before issuing a 'Return from Interrupt' instruction).

4.1.3. Thread status bits - TXSTATUS

Address: 04800010
Reset Value: 0x00020000 (thread 0), 0x00000000 (other threads)
Write Privilege: Variable

Note: For details of TXSTATUS in an FPU enabled core see the Meta FPU TRM.

Bit	Symbol	Description
31:24	Reserved	These bits are always 0.
23	CB1Marker	This bit is set to 1 when the catch state is caused to store data for interrupt level (ISTAT=1) and is otherwise similar to the CBMarker bit.
22	CBMarker	This bit is set to 1 when the catch state is caused to store data (at any ISTAT level). See the description of control unit registers 4.1.7 Catch state register 0 - TXCATCH0.

Bit	Symbol	Description
21:20	FReason	Memory fault reason: 00 - No error, 01 - General or Privilege violation, 10 - Page fault, 11 - Read Only Protection violation. This field will be set in conjunction with an instruction or data fetch memory fault in HReason
19:18	HReason	HALT trigger reason: 00 - SWITCH instruction, 01 - Instruction fetch memory fault or unknown instruction. 10 - Privilege violation * 11 - Data fetch memory fault. For code breakpoints HReason will be 01” and FReason will be 11. For data watchpoints HReason will be 11 and FReason will be 00. * Except for privilege violations due to accessing a region of memory without the required privilege which are shown as an Instruction or Data fetch memory fault.
17 [WRITE RESTRICTED]	PSTAT	This bit is set to 1 when the thread is in privileged mode. Writes to this bit will only affect the thread’s PSTAT register if the thread is turned off and the access has the necessary privilege (see below).
16 [WRITE RESTRICTED]	ISTAT	Indicates if the thread in an interrupt state. If this field is 0 we are in normal (background) mode, otherwise when 1 we are in an interrupt handler state. Writes to this bit will only affect the thread’s ISTAT register if the thread is turned off and the access has the necessary privilege (see below).
15:11 [READ ONLY]	Reserved	These bits are always '0'.
10:8	LSM_STEP	Indicates the current step of a load/store multiple instruction or pipelined memory-to-memory transfer instruction. This is required so that multi-issue load/store multiple instructions or the memory transfer instruction can be interrupted and then resumed.
7:5 [READ ONLY]	Reserved	These bits are always '0'.
4	SCC	Identifies if the captured condition flags were captured in a split 16x16 pipeline. This bit is 1 if the condition flags are derived from split arithmetic (therefore using the split arithmetic condition code table).
3	CF_Z/SCF_LZ	The state of this thread’s zero (Z) condition flag. If the flags were captured in split arithmetic mode this bit holds the zero flag from the low half word (LZ).
2	CF_N/SCF_HZ	The state of this thread’s negative (N) condition flag. If the flags were captured in split arithmetic mode this bit holds the zero flag from the high half word (HZ).

Bit	Symbol	Description
1	CF_V/SCF_HC	The state of this thread's overflow (V) condition flag. If the flags were captured in split arithmetic mode this bit holds the carry flag from the high half word (HC).
0	CF_C/SCF_LC	The state of this thread's carry (C) condition flag. If the flags were captured in split arithmetic mode this bit holds the carry flag from the low half word (LC).

The bottom 5-bits of the writeable portion of the thread status register will be updated internally whenever condition code capture occurs (e.g. by executing a compare instruction). It is also possible to write to this register directly. If writes from external sources (e.g. JTAG) occur at the same time as an internal write is completing, the internal access will be the one that completes.

4.1.4. Repeat count - TXRPT

Address: 04800018
Reset Value: 0x00000000
Write Privilege: None

Bit	Symbol	Description
31:0	RptCount	Repeat counter used for repeating instructions. As any repeating instruction must run for at least one cycle this value has an implied pre-decrement by 1, which means that 0x00000000 indicates a repeat value of 1 while 0xFFFFFFFF indicates a repeat value of 2^{32}.

4.1.5. Background timer - TXTIMER

Address: 04800020
Reset Value: 0x00000000
Write Privilege: None

Bit	Symbol	Description
31:0	BTimer	The current background timer value. This is auto-incremented at the timer clock frequency. If the value rolls over (from 0xFFFFFFFF to 0x00000000) a timer trigger will be caused. This trigger can be used as a source of interlock (via the trigger unit). Each thread has its own background and interrupt timers.

To allow an accurate real time clock to be maintained it is essential that no timer increments are effectively lost due to an update to the timer being made very soon after a timer tick. To achieve this, it is recommended that the timer is updated via a SWAP instruction whenever the timer is to be used for a real time clock.

4.1.6. Interrupt timer - TXTIMERI

Address: 04800068
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:0	ITimer	The current interrupt timer value. This is auto-incremented at the timer clock frequency. If the value rolls over (from 0xFFFFFFFF to 0x00000000) a timer trigger will be caused that feeds the appropriate interrupt trigger. This trigger can be used to cause an interrupt (via the trigger unit). Each thread has its own interrupt and background timers.

To allow an accurate real time clock to be maintained it is essential that no timer increments are effectively lost due to an update to the timer being made very soon after a timer tick. To achieve this, it is recommended that the timer is updated via a SWAP instruction whenever the timer is to be used for a real time clock.

4.1.7. Catch state register 0 - TXCATCH0

Address: 04800080
Reset Value: 0x00000000
Write Privilege: Variable

Note: For details of data capture for FPU exceptions see the Meta FPU TRM.

Bit	Symbol	Description
31:27	ReqLdReg	For Loads this field records the destination target register, for pipelined reads or writes this field will be zero.
26:16	ReqLdDest	For Loads this field records the destination target units: bit 26: DU1DSP bit 25: DU0DSP bit 24: DaOpPaMe template table bit 23: Trigger bit 22: FPU bit 21: PC bit 20: AU1 bit 19: AU0 bit 18: DU1 bit 17: DU0 bit 16: CTRB For pipelined reads or writes this field will be zero.
15	WPStop	Watchpoint stop bit.
14	WPSet	Watchpoint match set, indicates if a data watchpoint match with a count of 255 has occurred. This bit will also be set if a fault for an unaligned memory access occurs.

Bit	Symbol	Description
13:12	WPState	Watchpoint state (1-bit per watchpoint). These bits will always be zero when a fault for an unaligned memory access has occurred. 00 – unaligned memory access 01 – Watchpoint 0 is active 10 – Watchpoint 1 is active 11 – Both watchpoint are active
11:10	ReqState	Memory fault reason: 00 - No error, 01 - General violation, 10 - Page fault, 11 - Protection violation. *Note: If bits 14:10 of this register are zero then any other state in this register can be ignored.*
9	ReqPriv	Keeps a copy of the requests privilege level. If a request is reissued the appropriate current privilege level will be used and ReqPriv is discarded.
8	ReqRnW	Was the request a read (RnW='1') or a write (RnW='0').
7:0	ReqSB	For writes this contains the write mask. For loads the bottom five bits of this field are: bit 4: 0 bit 3: LNKGET marker (set to 1 to replay load as LNKGET) bit 2: Pro-processing (set to 1 if du1/au1 is first load dest) bit 1: Transfer width bit 1 bit 0: Transfer width bit 0 Transfer width decoding are: "00" - 8-bits ('B') "01" - 16-bits ('W') "10" - 32-bits ('D') "11" - 64-bits ('L') For reads the bottom five bits contain the destination port (RA, RABX, etc.).

When the watchpoint features are employed bits 12, 13 and 14 may be set depending upon watchpoint state. WPState bits are set on any requests that matched the watchpoint criteria and were either issued when the watchpoint counter was 255 or were not issued at all. In addition, the WPSet bit will be set if a watchpoint was matched with a count of 255. When a catch state dump is restored where an entry has this WPSet bit set to 1 that request will be treated as a watchpoint match with a watchpoint count of 255. In this state the value of the watchpoint count(s) will not be incremented (as would be normally done by watchpoint matches). Any exceptions raised in response to a watchpoint as the primary reason will be recorded as a memory fault with a fault reason on "00".

Any other reason for a memory operation to be stored in the catch registers will be indicated by a non-zero state in the memory fault reason bits. If both these bits and the watchpoint related bits are zero then no transaction is currently stored and the only reason for the CBMarker bit to be set in the status register is that the read pipeline was not empty.

4.1.8. Catch state register 1 - TXCATCH1

Address: 04800088
Reset Value: 0x00000000
Write Privilege: Variable
Note: For details of data capture for FPU exceptions see the Meta FPU TRM.

Bit	Symbol	Description
31:0	ReqAddr	The captured address for a transaction that required state to be saved. State is saved in response to a memory fault being returned from the MMU or in response to a data watchpoint.

4.1.9. Catch state register 2 - TXCATCH2

Address: 04800090
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:0	ReqDataL	The captured bottom 32-bits of data for a transaction that required state to be saved. State is saved in response to a memory fault being returned from the MMU or in response to a data watchpoint.

4.1.10. Catch state register 3 - TXCATCH3

Address: 04800098
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:0	ReqDataH	The captured top 32-bits of data for a transaction that required state to be saved. State is saved in response to a memory fault being returned from the MMU or in response to a data watchpoint

4.1.11. Deferred interrupt control - TXDEFR

Address: 048000A0
Reset Value: 0x00000000
Write Privilege: Variable
Note: For details of data for FPU deferred interrupts see the Meta FPU TRM.

Bit	Symbol	Description
31	Deferred Bus Error State (Flag)	This bit replicates the bottom bit of the deferred bus error state so that any odd-numbered error states can be easily determined from the sign of data read from TXDEFR.
30	Deferred Bus Error State (Source)	This bit indicates whether it was a data memory request or an instruction memory request that caused the bus error to be returned. This bit will be ‘0’ for errors reported for data memory requests, and ‘1’ for errors reported for instruction memory requests.
29:24	Deferred Bus Error State	These bits hold the data returned from the external memory subsystem associated with deferred bus errors (or linked get and set). 0x01 Bus decode Error 0x02 Linked set succeeded 0x04 Linked set failed Other values are reserved or system specific. Bit 0 set implies an error. Bit 0 clear implies a notification or warning.
23	BusErr Trigger Statel	A '1' indicates that a bus response has been received
22 [READ ONLY]	Reserved	This bit is always '0'.
21:8 [READ ONLY]	Reserved	These bits are always '0'.
7	BusErr ICtrl	Bus error interrupt control. Set this to ‘1’ for bus errors to affect interrupt deferred trigger, ‘0’ for background.
6:0 [READ ONLY]	Reserved	These bits are always '0'.

Note: The top 16-bits of this register mirror those available for reads of the TXSTAT/TXPOLL* registers. These bits in this register may be written (subject to privilege requirements) to set or clear deferred error states.*

4.1.12. Timer/catch state control - TXDIVTIME

Address: 048000E0
Reset Value: 0x00000001
Write Privilege: Variable

Bit	Symbol	Description
31	RPDirty	This bit is set to 1 when a thread is interrupted or halted with a read pipeline that contains data. Further pipeline read addresses cannot be issued until it is cleared either explicitly or implicitly by a return from interrupt or resumption of the halted thread. In the dirty state pipeline read data can be extracted from the read pipeline without interlocks with zero being delivered if an empty pipeline is read.
30:29 [READ ONLY]	Reserved	These bits are always '0'.

Bit	Symbol	Description
28	Reserved	This bit is always '0'.
27:24 [READ ONLY]	IRQEnc	Encoding of highest priority interrupt. This field is updated whenever a blocking read of TXSTATI is performed. The encoding represents the bit position of the highest priority interrupt trigger bit, with bit 15 encoded as “1111” and bit 0 encoded as “0000” and so on. Trigger bits 15 to 4 (external triggers) have the highest precedence with 15 the highest priority out of this set and 4 the lowest priority out of this set. Trigger bit 1 (kicks) is the next most important, with trigger bit 0 (timer) having a lower priority than kicks . Next comes trigger bit 3 (deferred), followed last of all by bit 2 (halt) which has the lowest priority out of all of the triggers.
23:22 [READ ONLY]	Reserved	These bits are always '0'.
21:16 [READ ONLY]	RPMask	These bits indicate the number of data entries currently stored in the read pipeline. Each active slot is indicated by a bit set to 1 starting at the least significant end of this field.
15:8 [READ ONLY]	Reserved	These bits are always '0'.
7:0	TFCtrl	This register provides a control for this thread over the frequency of the background and interrupt timer frequency. The Meta core base timer frequency is nominally 1MHz, but may be altered under software control. This register allows each thread to further divide the common Meta timer clock before application to each thread's timers. Values of 1 to 255 divide the clock by that amount, while a value of 0 divides the clock by 256.

4.1.13. Privilege extensions/step - TXPRIVEXT

Address: 048000E8
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31	CP7Priv	This bit indicates if privilege is required to interact with coprocessor 7 (in either direction). A '0' indicates that the coprocessor can be read/written at any time, while a setting of '1' indicates that only privileged users will be allowed.
30	CP6Priv	Similar to CP7Priv but for coprocessor 6
29	CP5Priv	Similar to CP7Priv but for coprocessor 5
28	CP4Priv	Similar to CP7Priv but for coprocessor 4
27	CP3Priv	Similar to CP7Priv but for coprocessor 3
26	CP2Priv	Similar to CP7Priv but for coprocessor 2

Bit	Symbol	Description
25	CP1Priv	Similar to CP7Priv but for coprocessor 1
24	CP0Priv	Similar to CP7Priv but for coprocessor 0
23:20 [READ ONLY]	Reserved	These bits are always '0'.
19	BTimerPriv	This bit indicates if privilege is required to write the TXTIMER control register (CT.4). A '0' indicates that the register can be written at any time, while a setting of '1' indicates that only privileged accesses will be allowed and unprivileged writes will raise an exception.
18	TracePriv	This bit indicates if privilege is required to write to the trace system control registers TTEXEC, TTCTRL or GTEXEC. This bit must be set to '1' if a thread may only write to these registers when privileged (otherwise '0').
17	TrigPriv	This bit indicates if privilege is required to make use of the thread's trigger mechanism (either in background or interrupt level). This bit must be set to '1' if a thread may only read/write its trigger registers when it is privileged (otherwise '0').
16	GCRPriv	This bit indicates if privilege is required to make use of any of the global common registers (i.e. registers that are shared between threads). Unprivileged accesses by a thread when its GCRPriv bit is set will cause a HALT condition to be raised (for a privilege violation).
15	Reserved	This bit is always '0'.
14	ILockPriv	This bit indicates if privilege is required to use the lock instruction (op-code 0xA8). A '0' indicates that the instruction can be used at any time, while a setting of '1' indicates that only privileged users will be allowed.
13	TICICycPriv	This bit indicates if privilege is required to write the TXTACTCYC or TXIDLECYC control registers (CT.30 and CT.31). A '0' indicates that these registers can be written at any time, while a setting of '1' indicates that only privileged accesses will be allowed.
12	TCBPriv	This bit indicates if privilege is required to write to the TXDIVTIME (CT.28) control register. Setting this to '1' allows a thread to read the registers but will deny it the ability to write anything to it unless it is in a privileged state.
11	Reserved	

Bit	Symbol	Description
10	ITimerPriv	This bit indicates if privilege is required to write the TXTIMERI control register (CT.13). A '0' indicates that the register can be written at any time, while a setting of '1' indicates that only privileged accesses will be allowed.
9	TStatusPriv	This bit indicates if privilege is required to write the top 24-bits of the TXSTATUS control register (CT.2) or any of the TXCATCH0, TXCATCH1, TXCATCH2 or TXCATCH3 registers (CT.16–CT.19). A '0' indicates that the register can be written at any time, while a setting of '1' indicates that only privileged accesses will be allowed.
8	TEnWPriv	This bit indicates if privilege is required to write to the TXENABLE control register CT.0. Setting this to '1' allows a thread to read this register but will deny it the ability to write anything to it unless it is in a privileged state.
7 [WRITE RESTRICTED]	MinimEnable	Setting this bit to ‘1’ enables the support for the MiniM instruction set. Writes to this bit will only affect the thread’s MinimEnable state if the thread is turned off and the access has the necessary privilege.
6 [READ ONLY]	Reserved	These bits are always ‘0’.
5	StaticBccPred	Setting this bit to ‘1’ disables dynamic branch prediction and uses a static branch prediction rule instead. The static branch prediction rule is that backwards conditional branches are always taken, whereas forwards conditional branches are never taken. Unconditional branches and CALLR instructions are not affected by this control as unconditional branches are always predicted as taken.
4	UnalignedFault	Setting this bit to ‘1’ enables the faulting of unaligned memory accesses. Once set if an unaligned memory access is seen it will be turned into a watchpoint which triggers and stops but does not actually use either of the watchpoints. Unaligned faults have priority over watchpoints in cases where both features would be activated for one specific memory access.
3	Reserved	
2	Step	Setting this bit to '1' enables a thread's instruction stream to be single stepped (including running single steps of a repeating instruction). Single stepping works in conjunction with the thread enable for a thread. In effect, setting this bit and the thread enable bit will allow one instruction execution (of any type - background/interrupt/etc.) to occur before the thread enable bit is set back to '0'.

Bit	Symbol	Description
1	Reserved	
0	PToggle	Setting this bit to '1' will cause privilege level to be toggled as interrupt state is entered or exited. Setting this to '0' when running interrupt code will allow normal background code to acquire privilege following the return from interrupt. If this bit is set to '1' a thread must be privileged to read or write its interrupt PCX and trigger registers TXSTATI and TXMASKI. Lastly, if this bit is set to a '1' a thread must be privileged to write its TXDEFR register (principally to protect the setting of the ICtrl bits).

4.1.14. Thread issue cycles - TXTACTCYC

Address: 048000F0
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:24 [READ ONLY]	Reserved	These bits are always '0'.
23:0 [READ ONLY]	TActive	This counter increments once for every instruction issued on this thread. Writing to this register will clear its contents.

4.1.15. Core idle cycles - TXIDLECYC

Address: 048000F8
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:24 [READ ONLY]	Reserved	These bits are always '0'.
23:0 [READ ONLY]	CIdle	This counter increments once for every cycle on which no instruction was issued for any thread. Writing to this register will clear its contents.

Note: This register is common to all threads (i.e. occupies the same slot in every thread's register block).

4.2. Per-thread kicks and privilege control registers

These registers control various functions in the Meta core but they do not have direct access via the control unit. Addresses given in this section are for thread0. This block of registers is repeated at intervals of 0x1000 for each thread in multi-threaded cores.

4.2.1. Thread 0 background kick - T0KICK

Address: 04800800
Reset Value: 0x00000000
Write Privilege: Variable
A write to this register causes a background level kick event to be sent to thread 0.

Bit	Symbol	Description
31:16 [READ ONLY]	Reserved	These bits are always '0'.
15:0 [WRITE ONLY]	T0BK	Writing the unsigned integer value n to the bottom 16-bits of this register will cause n kicks to be accumulated by thread 0's 16-bit background kick accumulator. For read these bits are always read as 0 as this register is merely a conduit and has no associated store

4.2.2. Thread 0 interrupt kick - T0KICKI

Address: 04800808
Reset Value: 0x00000000
Write Privilege: Variable
A write to this register will cause an interrupt level kick event to be sent to thread 0.

Bit	Symbol	Description
31:16 [READ ONLY]	Reserved	These bits are always '0'.
15:0 [WRITE ONLY]	T0IK	Writing the unsigned integer value n to the bottom 16-bits of this register will cause n kicks to be accumulated by thread 0's 16-bit interrupt kick accumulator. For read these bits are always read as 0 as this register is merely a conduit and has no associated store.

4.2.3. Thread 0 AMA register 4 - T0AMAREG4

Address: 04800810
Reset Value: 0x000000FF
Write Privilege: Variable

Bit	Symbol	Description
31:30 [READ ONLY]	Reserved	These bits are always '0'.
29:8	AMA_PSize0	This is the maximum number of instructions that may be outstanding in thread 0's AMA pool.
7:0	AMA_PInc0	This is the average issue rate that is used by the instruction rate control mechanism. This value is added to thread 0's current pool count once every sixteen cycles. This value is in effect a fractional number (with the lower 4-bits being fraction bits) as the bottom 4-bits of the delay count are not used when evaluating the number of instruction issues a thread wants to make.

4.2.4. Thread 0 AMA register 5 - T0AMAREG5

Address: 04800818
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:27	PARB_AMA_DeCnt0	Starting bit position of the 4-bit slice from the 26-bit AMA delay count T0AMAREG1, used for thread memory request arbitration. Allowable range 0x0 to 0x17.
26:0 [READ ONLY]	AMA_PCnt0	This counter holds the average number of instruction issues that thread 0 would like to make. The bottom four bits are all fraction bits. This quantity is a signed number.

4.2.5. Thread 0 AMA register 6 - T0AMAREG6

Address: 04800820
Reset Value: 0x00000000
Write Privilege: Variable

Bit	Symbol	Description
31:29 [READ ONLY]	Reserved	These bits are always '0'.
28:24	PARB_AMA_DLineCnt0	Starting bit position of the 4-bit slice from the 22-bit AMA deadline count T0AMAREG6, used for thread memory request arbitration. Allowable range 0x0 to 0x10.
23:4	AMA_DLineDt0	This holds the DEADLINE_DEFAULT value that will be loaded into thread 0's DEADLINE_COUNT (AMA_DLineCnt) when thread 0 leaves a waiting state.
3:0 [READ ONLY]	Reserved	These bits are always '0'.

4.2.6. Thread 0 memory mapped privilege - T0PRIVCORE

Address: 04800828
Reset Value: 0x00000000
Write Privilege: Always

This register is a set of single bits that control the privilege required to access the following registers: T0KICK, T0KICK, T0AMAREG4, T0AMAREG5 and T0AMAREG6.

When a bit is set, the appropriate register may only be accessed in the privileged mode.

Bit	Symbol	Description
31:3 [READ ONLY]	Reserved	These bits are always '0'.
2	T0AMAPriv	This bit indicates the privilege required to access any of the thread 0 memory mapped AMA registers. If set to 1 only privileged accesses will be able to read/write thread 0's AMA register 4, 5 or 6.

Bit	Symbol	Description
1	T0IKPriv	This bit indicates the privilege required to access the Thread 0 Interrupt Kick register. If set to 1 only privileged accesses will be able to write to the interrupt-level kicker.
0	T0BKPriv	This bit indicates the privilege required to access the Thread 0 Background Kick register. If set to 1 only privileged accesses will be able to write to the background kicker.

4.3. Global code breakpoint and data watchpoint setup

4.3.1. Any thread code breakpoint 0 address - CODEB0ADDR

Address: 0480FF00
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31:2	IBK0Addr	Comparison address for match.
1:0 [READ ONLY]	Reserved	These bits are always '0'.

4.3.2. Any thread code breakpoint 0 control - CODEB0CTRL

Address: 0480FF08
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31	IBK0En	Enable breakpoint.
30:29 [READ ONLY]	Reserved	These bits are always '0'.
28	IBK0TOnly	Match against requests from the given thread only if this bit is set to 1 otherwise match against requests from any thread (if 0).
27:24 [READ ONLY]	Reserved	These bits are always '0'.
23:16	IBK0Count	Hit counter; increments each time there is a hit up until it reaches 255 at which point an exception or trigger will be caused.
15:2	IBK0Mask	This allows a range of addresses to be used for the breakpoint. The allowed range is from 4 bytes to 64Kb. To ignore a given address bit the relevant mask bit should be set to 1 - so, for example, to breakpoint in any given 64-bit memory word just bit 2 should be 1.
1:0	IBK0Thread	Indicates which thread must issue the address to obtain a match.

4.3.3. Any thread code breakpoint 1 address - CODEB1ADDR

Address: 0480FF10

As above for Any thread code breakpoint 1.

4.3.4. Any thread code breakpoint 1 control - CODEB1CTRL

Address: 0480FF18

As above for Any thread code breakpoint 1.

4.3.5. Any thread code breakpoint 2 address - CODEB2ADDR

Address: 0480FF20

As above for Any thread code breakpoint 2.

4.3.6. Any thread code breakpoint 2 control - CODEB2CTRL

Address: 0480FF28

As above for Any thread code breakpoint 2.

4.3.7. Any thread code breakpoint 3 address - CODEB3ADDR

Address: 0480FF30

As above for Any thread code breakpoint 3.

4.3.8. Any thread code breakpoint 3 control - CODEB3CTRL

Address: 0480FF38

As above for Any thread code breakpoint 3.

4.3.9. Any thread data watchpoint 0 address - DATAW0ADDR

Address: 0480FF40

Reset Value: 0x00000000

Write Privilege: Always

Bit	Symbol	Description
31:0	DWP0Addr	Comparison address for address match. The bottom three bits are only used for writes.

4.3.10. Any thread data watchpoint 0 control - DATAW0CTRL

Address: 0480FF48

Reset Value: 0x00000000

Write Privilege: Always

Bit	Symbol	Description
31	DWP0REn	Enable watchpoint for reads.
30	DWP0WEn	Enable watchpoint for writes.
29	DWP0NotT	If DWP0TOnly (see next) is set this bit can be used to invert the rule so that any thread other than DWP0Thread causes the watchpoint to trigger. If DWP0TOnly is not set this bit does nothing.

Bit	Symbol	Description
28	DWP0TOnly	Match against requests from the given thread only if this bit is set to 1, otherwise match against requests from any thread (if 0).
27:24	DWP0Size	Write size specifier: 0000 - any 0001 - byte only 0010 - 16-bit word only 0011 - 32-bit dword only 0100 - 64-bit lword only. If a byte, word, dword or lword write size is specified it is possible to employ the data masking and matching logic to test for specific write patterns.
23:16	DWP0Count	Hit counter; increments each time there is a hit. If a hit occurs when the count is 255 a memory fault will be reported for that access and no further reads or writes will be issued by the thread until the exception that may result has been handled.
15:3	DWP0Mask	This allows a range of addresses to be used for the watchpoint. The allowed range is from 8 bytes to 64Kb.To ignore a given address bit the relevant mask bit should be set to 1 - so, for example, to breakpoint in any given pair of 64-bit memory words just bit 3 should be 1.
2 [READ ONLY]	Reserved	This bit is always '0'.
1:0	DWP0Thread	Indicates which thread must issue the address to obtain a match.

4.3.11. Any thread data watchpoint 0 DataL - DATAW0DMATCH0

Address: 0480FF50
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31:0	DWP0DataL	Comparison data for write data match (low word).

Watchpoints allow matching of writes against specific patterns. When this facility is used DataL and DataH should contain the reference data to compare against, while MaskL and MaskH contain a bitwise mask to be applied to the write data before matching against the reference data.

Therefore, the test that is applied is effectively:

```
If ((WriteData AND Mask) == ReferenceData)
        WatchpointValid = TRUE;
```

If bytes, words or dwords are being matched against it will be necessary to set MaskH/DataH and potentially some of MaskL/DataL to zero to allow the unused byte lanes to be ignored. To disable data matching altogether MaskL, DataL, MaskH and DataH must all be set to zero.

The write data used for watchpoint comparison is always the unaligned form (i.e. the write data before manipulation to put the data in the right byte lanes based upon the address). In practical terms this means that byte transactions use the bottom 8-bits, words use the bottom 16-bits and so on.

DATAW0DMATCH0, DATAW0DMATCH1, DATAW0DMASK0 and DATAW0DMASK1 are not used for read watchpoints.

4.3.12. Any thread data watchpoint 0 DataH - DATAW0DMATCH1

Address: 0480FF58
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31:0	DWP0DataH	Comparison data for write data match (high word).

4.3.13. Any thread data watchpoint 0 MaskL - DATAW0DMASK0

Address: 0480FF60
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31:0	DWP0MaskL	Comparison mask for write data match (low word).

4.3.14. Any thread data watchpoint 0 MaskH - DATAW0DMASK1

Address: 0480FF68
Reset Value: 0x00000000
Write Privilege: Always

Bit	Symbol	Description
31:0	DWP0MaskH	Comparison mask for write data match (high word).

4.3.15. Any thread data watchpoint 1 address - DATAW1ADDR

Address: 0480FF80
As above for Any thread data watchpoint 0.

4.3.16. Any thread data watchpoint 1 control - DATAW1CTRL

Address: 0480FF88
As above for Any thread data watchpoint 0.

4.3.17. Any thread data watchpoint 1 DataL - DATAW1DMATCH0

Address: 0480FF90
As above for Any thread data watchpoint 0.

4.3.18. Any thread data watchpoint 1 DataH - DATAW1DMATCH1

Address: 0480FF98
As above for Any thread data watchpoint 0.

4.3.19. Any thread data watchpoint 1 MaskL - DATAW1DMASK0

Address: 0480FFA0

As above for Any thread data watchpoint 0.

4.3.20. Any thread data watchpoint 1 MaskH - DATAW1DMASK1

Address: 0480FFA8

As above for Any thread data watchpoint 0.

4.3.21. Internal core events 0 - PERF_ICORE0

Address Offset: 0x0480FFD0

Reset Value: 0x00000000

Write Privilege: Always

This control allows additional performance metrics to be captured in the Meta HTP core (via performance counter 0).

Bit	Symbol	Description
31:10 [READ ONLY]	Reserved	These bits are always '0'.
9:8	PerfICore0PrivFilter	Privilege Filter. If set to: 00 - all relevant performance counter events are counted in PERF_COUNT0 01 - reserved 10 - only materials for events caused in unprivileged state are counted in PERF_COUNT0. 11 - only materials for events caused in privileged state are counted in PERF_COUNT0
7:4 [READ ONLY]	Reserved	These bits are always '0'.
3:0	ICoreCtrl	Internal core event counter control specifier: 0000 - Count Dcache read TLB hit and cache misses 0001 - Count Icache TLB hit and cache misses 0010 - Count Dcache TLB misses 0011 - Count Icache TLB misses 0100 – Count Dcache write TLB hits 0101 – Count Dcache write TLB misses 0110 - reserved (no operation) 0111 - reserved (no operation) 1000 - Count Dcache read cache line fetch 1001 - Count Icache read cache line fetch 1010 - Count Dcache read single word fetch 1011 - Count Icache read single word fetch 1100 - reserved (no operation) 1101 - reserved (no operation) 1110 - Count memory writes stalled at point of core issue 1111 – reserved (no operation)

4.3.22. Internal core events 1 - PERF_ICORE1

Address Offset: 0x0480FFD8

As above for PERF_ICORE0

4.3.23. Performance counter 0 - PERF_COUNT0

Address Offset: 0x0480FFE0

Reset Value: 0x0F000000

Write Privilege: Always

The performance counters allow the occurrence of certain types of performance-related events to be monitored for one or more threads as code executes.

Bit	Symbol	Description
31:28	Ctrl	Counter control specifier: 0000 - Count cycles with superthreads 0001 - Count instruction issue rewinds caused by dcache misses 0010 - Count cycles with rewinds and superthreads (for all threads, mask ignored) 0011 - Count cycles since execution start (for all threads, mask ignored) 0100 - count all predicted conditional branches 0101 - count all miss-predicted conditional branches 0110 - count all predicted function returns 0111 - count all miss-predicted function returns 1000 - Count Dcache hits 1001 - Count Icache hits 1010 - Count Icache misses 1011 - Count Dcache_MMU request stalled 1100 - Count Icache_MMU request stalled 1101 - Internal Core Events (selected by the PERF_ICORE0 register) 1110 – reserved (no operation) 1111 – Count external events selected by performance channel 0 register.(extended, see below)
27:24	ThreadMask	Enables/disables performance counting for each thread. If a bit is set to 1, events will be counted for corresponding thread. Bit 24 for T0 etc.
23:0	Count	Event counter; increments each time an event of type specified by Ctrl field occurs for any of the threads specified in thread mask

The conditional branch and function return prediction counting provides two options: firstly it is possible to form a count of all predictions (where the predictions are predicted to be taken or not taken), secondly it is possible to count the number of times such predictions are made in error. With one counter tracking the first of these metrics and the other counter tracking the second it is possible to determine an overall picture of the efficacy of the Meta HTP prediction and speculative execution changes.

Note: Ctrl settings 0010 and 0011 are not masked on a per-thread basis and are therefore also not filtered for privilege when PerfICore0PrivFilter is set. Otherwise, PerfICore0PrivFilter is used for all Ctrl settings except 1111 for which PerfChan0PrivFilter is used instead.

4.3.24. Performance counter 1 - PERF_COUNT1

Address Offset: 0480FFE8

Reset Value: 0x1F000000

As above for PERF_COUNT0

4.3.25. Performance channel 0 - PERF_CHAN0

Address Offset: 0x04830150 (0x03000050)

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Note: Support of this register at location of 0x03000050 will become obsolete in a future version.

The performance counters allow the occurrence of certain types of performance-related events to be monitored for one or more threads as code executes. This register provides the routing for further events to be channelled into the PERF_COUNT0 register. For monitoring write combiner, pre-arbiter and system port events, the numbers recorded in the PERF_COUNT0 register may be inaccurate if the core clock is slower than the system clock. When the core clock is faster than the system clock, the number in the PERF_COUNT0 register should be divided by their frequency ratio.

Bit	Symbol	Description
31:10	Reserved	These bits are always '0'.
9:8	PerfICore0PrivFilter	Privilege Filter. If set to: 00 - all relevant performance counter events are counted in PERF_COUNT0. 01 - reserved. 10 - only materials for events caused in unprivileged state are counted in PERF_COUNT0. 11 - only materials for events caused in privileged state are counted in PERF_COUNT0.
7:4 [READ ONLY]	Reserved	These bits are always '0'.
3:0	Channel	0000 - write combiner output write bursts 0001 - write combiner output writes 0010 - write combiner output read bursts 0011 - write combiner output reads 0100 - pre-arbiter port stalls 0101 - core memory request stalls at cross bar input 0110 - Reserved 0111 - Reserved 1000 - Reserved 1001 - Meta core register request stalls at cross-bar input 1010 - Reserved 1011 - Meta core register port stalls 1100 - Core memory port stalls 1101 - write combiner port stalls (all reasons) 1110 - write combiner stalls due to flushes 1111 - Master system port stalls

4.3.26. Performance channel 1 - PERF_CHAN1

Address Offset: 0x04830158 (0x03000058)

As above for PERF_CHAN0

4.4. Write combiner configuration registers

Note: Not all cores are fitted with write combiners, refer to the relevant Core.Configuration Specification document supplied for details.

4.4.1. Write combiner config register 0 - WRCOMBCONFIG0

Address Offset: 0x04830100 (0x03000000)
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG
Note: Support of this register at location of 0x03000000 will become obsolete in a future version.

Bit	Symbol	Description
31:14	Reserved	
13	TxWrCombEn	Set to 1 to enable combining/bursting. Note MMU must also be switched on and the write combine bit in the appropriate MMU table entry for the current write address must also be set.
12	TxWrCombTO	Set to 1 to enable write combiner timeout. With timeout enabled the write combiner will automatically flush after TxWrCombTOCnt clock cycles after the write combiner starts to fill.
9:0	TxWrCombTOCnt	Timeout count value in core clock cycles.

4.4.2. Write combiner config register 1 - WRCOMBCONFIG1

Address Offset: 0x04830108 (0x03000008)
This register has the same contents as WRCOMBCONFIG0 but for thread1

4.4.3. Write combiner config register 2 - WRCOMBCONFIG2

Address Offset: 0x04830110 (0x03000010)
This register has the same contents as WRCOMBCONFIG0 but for thread2

4.4.4. Write combiner config register 3 - WRCOMBCONFIG3

Address Offset: 0x04830118 (0x03000018)
This register has the same contents as WRCOMBCONFIG0 but for thread3

4.5. Privilege registers

The Meta core's global privilege registers consist of a set of bit-masks that restrict access to parts of the memory map to threads executing at privileged level.

Initially the processor executes its first thread in a privileged mode with a minimal level of privilege protection enabled. This initial boot task may remove any restriction before becoming unprivileged, or

may force a much higher level of protection, therefore retaining privileged access to selected parts of the system.

These following registers each specify privilege related behaviour delivered to the corresponding thread when it accesses the corresponding part of the system region:

4.5.1. System region privilege for Thread 0 - T0PRIVSYSR

Address: 04810000

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:5	Reserved	
4	Core Code Memory 80000000- 81FFFFFF	0 - No restriction 1 - No unprivileged operation allowed
3	Core Cached Memory 84000000- 87FFFFFF	0 - No restriction 1 - No unprivileged operation allowed
2	Direct mapped 06000000- 07FFFFFF	0 - Writes unrestricted. Reads fail. 1 - No unprivileged writes. Reads fail.
1	Reserved	
0	Cache Flush 04400000- 047FFFFF	0 - No restriction. 1 - No unprivileged operation allowed.

4.5.2. System region privilege for Thread 1 - T1PRIVSYSR

Address: 04810008

Reset Value: 0x00000000

This register has the same contents as T0PRIVSYSR but for thread1

4.5.3. System region privilege for Thread 2 - T2PRIVSYSR

Address: 04810010

Reset Value: 0x00000000

This register has the same contents as T0PRIVSYSR but for thread2

4.5.4. System region privilege for Thread 3 - T3PRIVSYSR

Address: 04810018

Reset Value: 0x00000000

This register has the same contents as T0PRIVSYSR but for thread3

4.5.5. Core and expansion privilege for Thread 0 - T0PIOREG

Address: 04810100

Reset Value: 0x0000000B

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:16	Priv_EXP	A bit set prevents non-privileged threads from accessing the corresponding region in the Custom area 02000000-02FFFFFF.
15:0	Priv_META_Core_Registers	A bit set prevents non-privileged threads from accessing the corresponding 64kbyte region in the register region 04800000-048FFFFF.

4.5.6. Core and expansion privilege for Thread 1 - T1PIOREG

Address: 04810108

Reset Value: 0x0000000B

This register has the same contents as T0PIOREG but for thread1

4.5.7. Core and expansion privilege for Thread 2 - T2PIOREG

Address: 04810110

Reset Value: 0x0000000B

This register has the same contents as T0PIOREG but for thread2

4.5.8. Core and expansion privilege for Thread 3 - T3PIOREG

Address: 04810118

Reset Value: 0x0000000B

This register has the same contents as T0PIOREG but for thread3

4.5.9. System event privilege control for Thread 0 - T0PSYREG

Address: 04810180

Reset Value: 0xFFFFFFE0

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	T0PSYREG	A bit set prevents non-privileged thread from generating the corresponding pair of System Events via addresses in each 128-byte region in the range 04000000-04000FFF

4.5.10. System event privilege control for Thread 1 - T1PSYREG

Address: 04810188

Reset Value: 0xFFFFFFE0

This register has the same contents as T0PSYSREG but for thread1

4.5.11. System event privilege control for Thread 2 - T2PSYREG

Address: 04810190

Reset Value: 0xFFFFFFE0

This register has the same contents as T0PSYSREG but for thread2

4.5.12. System event privilege control for Thread 3 - T2PSYREG

Address: 04810198

Reset Value: 0xFFFFFFE0

This register has the same contents as T0PSYSREG but for thread3

4.6. Trigger control registers

See section 2.4 for a detailed description of Meta triggers.

HWSTATMETA, HWSTATEXT, HWSTATEXT* - edge-triggered model

A hardware trigger edge causes the corresponding status/clear register bit to be set to one and a corresponding trigger edge is sent to the Meta core. Multiple triggers leave the bit set at one.

Writing a one to any bit in the status/clear register at any time causes the bit concerned to toggle from either one to zero or zero to one. If this software event is combined exactly with the occurrence of a significant hardware trigger edge then the resulting state of the status/clear bit will always be one.

If one is written to a status/clear register bit which is previously zero a software generated trigger edge will be sent to the Meta core if the corresponding trigger vector is set.

HWSTATEXT, HWSTATEXT*- level-sensitive model

The status read from the status/clear register will indicate directly the state of the trigger signal produced by the hardware source. Each time the hardware trigger signal changes from zero to one an event will be generated for the Meta core provided the corresponding trigger vector is set.

If software writes a one to a status/clear register bit a corresponding trigger will be sent to be Meta core provided the corresponding trigger vector is set.

TnVECINT_*, TnVECEXT2, and TnVECEXT - trigger vectors

A zero in any of the 4-bit fields within these registers disables delivery of the trigger concerned. The values 4 to 15 'vectors' the triggers to the corresponding H/W trigger bits of the trigger status registers within the Meta core.

By convention H/W trigger 15 is the highest priority trigger. Normally fewer trigger priority levels than the maximum of twelve supported are required. Conventionally H/W triggers 4 to 7 are used for trigger level 1 triggers handled separately on each of four threads by background level handlers. Also H/W triggers 8 to 11 can be used as level 2 triggers for these threads via interrupt level handlers. The remaining H/W triggers 12 to 15 can be used to implement global trigger levels 3 to 6 respectively within nestable interrupt level handlers on any chosen thread.

Placing the value 1 in any trigger vector allows the trigger to be detected by the Debug Port, this allows an external debugger to detect threads that breakpoint or HALT.

Placing the value 2 or 3 in any trigger vector allows the trigger to be detected by the Slave Port. This allows an external Host to detect threads that breakpoint or HALT.

4.6.1. Hardware trigger status META - HWSTATMETA

Address: 04820000

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:18	Reserved	
17	PERF1TRIG	A status/clear bit describing the occurrence of a roll-over event in the PERF_COUNT1 register (where a roll-over is a transition from 0xFFFFFF to 0x000000).

Bit	Symbol	Description
16	PERF0TRIG	A status/clear bit describing the occurrence of a roll-over event in the PERF_COUNT0 register (where a roll-over is a transition from 0xFFFFFF to 0x000000).
15	Reserved	
14	T3PTRIG	A status/clear bit describing the occurrence of memory page fault from Meta thread 3
13	T3ITRIG	A status/clear bit describing the occurrence of interrupt trigger from Meta thread 3
12	T3BTRIG	A status/clear bit describing the occurrence of background trigger from Meta thread 3
11	Reserved	
10	T2PTRIG	A status/clear bit describing the occurrence of memory page fault from Meta thread 2
9	T2ITRIG	A status/clear bit describing the occurrence of interrupt trigger from Meta thread 2
8	T2BTRIG	A status/clear bit describing the occurrence of background trigger from Meta thread 2
7	Reserved	
6	T1PTRIG	A status/clear bit describing the occurrence of memory page fault from Meta thread 1
5	T1ITRIG	A status/clear bit describing the occurrence of interrupt trigger from Meta thread 1
4	T1BTRIG	A status/clear bit describing the occurrence of background trigger from Meta thread 1
3	Reserved	
2	T0PTRIG	A status/clear bit describing the occurrence of memory page fault from Meta thread 0
1	T0ITRIG	A status/clear bit describing the occurrence of interrupt trigger from Meta thread 0
0	T0BTRIG	A status/clear bit describing the occurrence of background trigger from Meta thread 0

All of these triggers use the edge-triggered model.

4.6.2. Hardware trigger status 0-31 - HWSTATEXT

Address: 04820010
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWSTATEXT	A status/clear register describing the occurrence of external triggers 0-31.

4.6.3. Hardware trigger status 32-63 - HWSTATEXT2

Address: 04820018
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWSTATEXT2	A status/clear register describing the occurrence of external triggers 32-63.

4.6.4. Hardware trigger status 64-95 - HWSTATEXT4

Address: 04820020
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWSTATEXT4	A status/clear register describing the occurrence of external triggers 64-95.

4.6.5. Hardware trigger status 96-128 - HWSTATEXT6

Address: 04820028
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWSTATEXT6	A status/clear register describing the occurrence of external triggers 96-128.

4.6.6. Hardware trigger edge/level configuration - HWLEVELEXT

Address: 04820030
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWLEVELEXT	A read/write register which defines the edge/level behaviour of HWSTATEXT. If a bit in this register is 1, the related trigger obeys the level-sensitive trigger model, and if 0 the edge-triggered model.

4.6.7. Hardware trigger edge/level configuration 2 - HWLEVELEXT2

Address: 04820038
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWLEVELEXT2	A read/write register which defines the edge/level behaviour of HWSTATEXT2. If a bit in this register is 1, the related trigger obeys the level-sensitive trigger model, and if 0 the edge-triggered model.

4.6.8. Hardware trigger edge/level configuration 4 - HWLEVELEXT4

Address: 04820040
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWLEVELEXT4	A read/write register which defines the edge/level behaviour of HWSTATEXT4. If a bit in this register is 1, the related trigger obeys the level-sensitive trigger model, and if 0 the edge-triggered model.

4.6.9. Hardware trigger edge/level configuration 6 - HWLEVELEXT6

Address: 04820048
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWLEVELEXT6	A read/write register which defines the edge/level behaviour of HWSTATEXT6. If a bit in this register is 1, the related trigger obeys the level-sensitive trigger model, and if 0 the edge-triggered model.

4.6.10. Hardware trigger mask - HWMASKEXT

Address: 04820050
Reset Value: 0xFFFFFFFF
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	HWMASKEXT	A register to enable/disable external triggers 0-31. If a bit in this register is 1, the related trigger is enabled.

4.6.11. Hardware trigger mask 2 - HWMASKEXT2

Address: 04820058

Reset Value: 0xFFFFFFFF

Write Privilege: Controlled by TxPIOREG . If external triggers 32-63 does not exist, this register would be removed and returns 0x0.

Bit	Symbol	Description
31:0	HWMASKEXT2	A register to enable/disable external triggers 32-63. If a bit in this register is 1, the related trigger is enabled.

4.6.12. Hardware trigger mask 4 - HWMASKEXT4

Address: 04820060

Reset Value: 0xFFFFFFFF

Write Privilege: Controlled by TxPIOREG . If external triggers 64 - 95 does not exist, this register would be removed and returns 0x0.

Bit	Symbol	Description
31:0	HWMASKEXT4	A register to enable/disable external triggers 64-95. If a bit in this register is 1, the related trigger is enabled.

4.6.13. Hardware trigger mask 6 - HWMASKEXT6

Address: 04820068

Reset Value: 0xFFFFFFFF

Write Privilege: Controlled by TxPIOREG . If external triggers 96 - 127 does not exist, this register would be removed and returns 0x0.

Bit	Symbol	Description
31:0	HWMASKEXT6	A register to enable/disable external triggers 96-127. If a bit in this register is 1, the related trigger is enabled.

4.6.14. Thread0 background trigger vector - T0VECINT_BHALT

Address: 04820500

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	BHALT	A read/write register containing a 4-bit trigger vector value for the background trigger generated by the Meta core related to the state of thread 0.

4.6.15. Thread0 interrupt trigger vector - T0VECINT_IHALT

Address: 04820508

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description

Bit	Symbol	Description
3:0	IHALT	A read/write register containing a 4-bit trigger vector value for the interrupt trigger generated by the Meta core related to the state of thread 0.

4.6.16. Thread0 memory fault trigger vector - T0VECINT_PHALT

Address: 04820510
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	PHALT	A read/write register containing a 4-bit trigger vector value for the memory read fault trigger generated by the Meta core related to the state of thread 0.

4.6.17. Thread1 background trigger vector - T1VECINT_BHALT

Address: 04820520
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	BHALT	A read/write register containing a 4-bit trigger vector value for the background trigger generated by the Meta core related to the state of thread 1.

4.6.18. Thread1 interrupt trigger vector - T1VECINT_IHALT

Address: 04820528
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	IHALT	A read/write register containing a 4-bit trigger vector value for the interrupt trigger generated by the Meta core related to the state of thread 1.

4.6.19. Thread1 memory fault trigger vector - T1VECINT_PHALT

Address: 04820530
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	PHALT	A read/write register containing a 4-bit trigger vector value for the memory read fault trigger generated by the Meta core related to the state of thread 1.

4.6.20. Thread2 background trigger vector - T2VECINT_BHALT

Address: 04820540
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	BHALT	A read/write register containing a 4-bit trigger vector value for the background trigger generated by the Meta core related to the state of thread 2.

4.6.21. Thread2 interrupt trigger vector - T2VECINT_IHALT

Address: 04820548
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	IHALT	A read/write register containing a 4-bit trigger vector value for the interrupt trigger generated by the Meta core related to the state of thread 2.

4.6.22. Thread2 memory fault trigger vector - T2VECINT_PHALT

Address: 04820550
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	PHALT	A read/write register containing a 4-bit trigger vector value for the memory read fault trigger generated by the Meta core related to the state of thread .

4.6.23. Thread3 background trigger vector - T3VECINT_BHALT

Address: 04820560
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	BHALT	A read/write register containing a 4-bit trigger vector value for the background trigger generated by the Meta core related to the state of thread 3.

4.6.24. Thread3 interrupt trigger vector - T3VECINT_IHALT

Address: 04820568
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	IHALT	A read/write register containing a 4-bit trigger vector value for the interrupt trigger generated by the Meta core related to the state of thread 3.

4.6.25. Thread3 memory fault trigger vector - T3VECINT_PHALT

Address: 04820570
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	PHALT	A read/write register containing a 4-bit trigger vector value for the memory read fault trigger generated by the Meta core related to the state of thread 3.

4.6.26. PERF0 trigger vector – PERF0VECINT

Address: 04820580
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	PERF0TR	A read/write register containing a 4-bit trigger vector value for the PERF0TRIG generated by the Meta core.

4.6.27. PERF1 trigger vector – PERF1VECINT

Address: 04820588
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:0	PERF1TR	A read/write register containing a 4-bit trigger vector value for the PERF1TRIG generated by the Meta core.

4.6.28. External hardware trigger vector table 0 - HWVEC0EXT

Address: 04820700-048207F8
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	HWVEC0EXT	A table of up to 32 4-bit read/write trigger vector values - one in each 64-bit location. These are related to each co-processor or external hardware trigger source in a system dependent fashion.

4.6.29. External hardware trigger vector table 2 - HWVEC20EXT

Address: 04821700-048217F8
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	HWVEC20EXT	A table of up to 32 4-bit read/write trigger vector values - one in each 64-bit location. These are related to each co-processor or external hardware trigger source in a system dependent fashion.

4.6.30. External hardware trigger vector table 4 - HWVEC40EXT

Address: 04822700-048227F8
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	HWVEC04EXT	A table of up to 32 4-bit read/write trigger vector values - one in each 64-bit location. These are related to each co-processor or external hardware trigger source in a system dependent fashion.

4.6.31. External hardware trigger vector table 6 - HWVEC60EXT

Address: 04823700-048237F8
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
3:0	HWVEC60EXT	A table of up to 32 4-bit read/write trigger vector values - one in each 64-bit location. These are related to each co-processor or external hardware trigger source in a system dependent fashion.

4.7. General Meta control registers

This region contains registers that have many independent purposes.

4.7.1. META core ID - METAC_ID

Address: 04830000
Reset Value: Depends on core revision
Write Privilege: Read only
This register is read-only and contains a predefined value to identify the Meta core used.

Bit	Symbol	Description
31:24	METAC_ID_MAJOR_BITS	Meta core Major identifier (02)
23:16	METAC_ID_MINOR_BITS	Meta core Minor identifier (01)
15:8	METAC_ID_REV_BITS	Meta core Revision identifier (03/04)

Bit	Symbol	Description
7:0	METAC_ID_MAINT_BITS	Meta core Maintenance revision identifier (XX)

Current versions of Meta HTP have major, minor and rev bits set to 2.1.3 or 2.1.4. The 3rd digit (METAC_ID_REV_BITS) is actually used to indicate the instruction set present. The 4 vs. 3 indicates the presence or not of a floating point unit (FPU).

4.7.2. Meta core configuration ID - CORE_ID

Address: 0x04831000

Reset Value: 0x140X0000

Write Privilege: Read only

This register is read-only and contains a predefined value to identify the type and configuration of Meta core used.

Bit	Symbol	Description
31:24	IMG_META_GROUP_ID	Set to 0x14 to indicate that it is a Meta CPU/DSP core.
23:16	IMG_META_CORE_ID	This field is set to 0x0X for Meta HTP core configurations and 0x1X for Meta MTP and LTP cores.
15:0	IMG_META_CONFIG	2:0 CACHE_TYPE (0=FULLMMU, 1=NOMMU, 2=NOCACHE, 3=PRIVNOMMU, 4:7=reserved) 5:3 DSP_TYPE (0=Extended, 1=Standard, 2:7=reserved) 6 Reserved 8:7 FPU_TYPE (0=None, 1=single precision, 2=double precision, 3=reserved) 10 GLOB_CACHE_COH (0=Included, 1=not present) 15:11 Reserved

4.7.3. Meta core revision - CORE_REV

Address: 0x04831008

Reset Value: 0x000201XX

Bit	Symbol	Description
31:24	IMG_META_DESIGNER	Designer Code
23:16	IMG_META_MAJOR_REV	IMG Core Major Revision
15:8	IMG_META_MINOR_REV	IMG Core Minor Revision
7:0	IMG_META_MAINT_REV	IMG Core Maintenance Revision

IMG_META_MAJOR_REV, IMG_META_MINOR_REV and IMG_META_MAINT_REV are identical to corresponding fields in METAC_ID Register. The METAC_ID register is kept in place to maintain backwards compatibility with software and tools but may be obsoleted in future.

The DESIGNER field will normally be set to 0x00. In some circumstances it may be changed to a different value for a customer specific release.

4.7.4. Meta core configuration ID 2 - CORE_CONFIG2

Address 0x04831020

Reset Value: Depends on Meta core type

Write Privilege: Read only

A 32-bit read only register identifying core specific configuration details

Bit	Symbol	Description
31	BASELINE_DSP	When set indicates that the DSP per-thread register set of 9 registers per data unit (D0.0-D0.8+D1.0-D1.8) is used (as opposed to 8 registers per data unit for general purpose threads and 16 registers per data unit for extended DSP threads). See the thread capabilities table in Section 4.1.1 for details.
30:29	CORE_DEBUG_TYPE	Indicates the features available to the debug port: when set to "0" debug can be routed through the core or cache backend and the MCM* registers are supported; when set to "1" debug can only be routed through the cache backend and the MCM* registers are supported (principally to allow access to core memories); lastly when set to "2" debug is routed as if it always goes through core and the MCM* registers are not required (or available) for access to core memories.
28	SEG_MMU	When set indicates that Segment MMU is included. When unset indicates that Segment MMU is not included.
27	DCACHE_WB	When set indicates that the DCACHE operates in write back-mode. When unset indicates that the DCACHE operates in write-through mode.
26	DCACHE_SMALL	When set indicates that the DCACHE is small (2k or less)
25	ICACHE_SMALL	When set indicates that the ICACHE is small (2k or less)
24:22	DCACHE_SIZE_NP	This field is the difference between data cache size as described by field DCACHE_SIZE and the actual data cache size. It is in units of 1/16th of data cache size as described by field DCACHE_SIZE. 0 – no difference 1:7 - Reserved
21:19	ICACHE_SIZE_NP	This field is the difference between instruction cache size as described by field ICACHE_SIZE and the actual instruction cache size. It is in units of 1/16th of instruction cache size as described by field ICACHE_SIZE. 0 – no difference 1:5 - Reserved 6 – difference of 6/16th 7 - Reserved

Bit	Symbol	Description
18:16	DCACHE_SIZE	This field describes the data cache size rounded up to the nearest power of 2. N.B. use in conjunction with the DCACHE_SMALL bit (above) 0 – 4k / 64bytes 1 – 8k / 128 bytes 2 – 16k / 256 bytes 3 – 32k / 512 bytes 4 – 64k /1024 bytes 5 – 128k /2048 bytes 6:7 - Reserved
15:13	ICACHE_SIZE	This field describes the instruction cache size rounded up to the nearest power of 2. N.B. use in conjunction with the DCACHE_SMALL bit (above) 0 – 4k / 64 bytes 1 – 8k / 128 bytes 2 – 16k / 256 bytes 3 – 32k / 512 bytes 4 – 64k /1024 bytes 5 – 128k /2048 bytes 6:7 - Reserved
12:11	GLOBAL_ACCUMULATORS	0 – No accumulators 1 – 1 global accumulator per data unit 2 – 2 global accumulators per data unit 3 – 3 global accumulators per data unit
10:8	GLOBAL_DU_REGISTERS	0 – No Global registers per data unit 1 – 1 Global registers per data unit 2 – 2 Global registers per data unit 3 – 4 Global registers per data unit 4 – 8 Global registers per data unit 5 – 16 Global registers per data unit 6:7 Reserved
7:5	GLOBAL_AU_REGISTERS	0 – No Global registers per addr unit 1 – 1 Global registers per addr unit 2 – 2 Global registers per addr unit 3 – 4 Global registers per addr unit 4 – 8 Global registers per addr unit 5:7 Reserved
4	REAL_TIME_TRACE	0 – Real time trace present 1 – Real time trace not present
3:2	WATCHPOINTS	0 – No Data Watchpoints available 1 – 2 Data Watchpoints available 2,3 – Reserved

Bit	Symbol	Description
1:0	BREAKPOINTS	0 – No Breakpoints available 1 – 2 Breakpoints available 2 – 4 Breakpoints available 3 – Reserved

4.7.5. MMU table base - MMCU_TABLE_PHYS

Address: 04830010

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

The TABLE_PHYS registers must be set before exiting MMU bypass mode. DCACHE_FLUSH and ICACHE FLUSH registers can then be used to initiate and wait for the caches to be flushed. Then the DCACHE_CTRL and ICACHE_CTRL registers can be set to enable normal cache operation. The MMU cache must be explicitly flushed after reset before it is first actually used (see section 3.2.4 Cache / TLB invalidate).

Bit	Symbol	Description
31:2	Base	Physical base address of “root” table N.B. This is not used for the Meta HTP enhanced MMU table structure
1	Reserved	
0	Mode	0 = Meta ATP MMU table structure(default) 1 = Meta HTP enhanced MMU table structure

Note: Compatibility with Meta ATP can be achieved by setting bit 0 to ‘0’ or by setting up the new MMCU_TnLOCAL_TABLE_PHYS registers to appropriate values.*

4.7.6. Tn local range root table 0 - MMCU_TnLOCAL_TABLE_PHYS0

Address: 0x04830700 + 0x20*n

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Tn Local Range Root Table Register0 (see also section 2.7)

Bit	Symbol	Description
31:22	Base	Bit 31 to 22 of the Linear Base Address for thread n. The top bit must always be set to zero.
21:12	Reserved	

Bit	Symbol	Description
11:8	Range	Range of Linear Address 0 = 4M 1 = 8M 2 = 16M 3 = 32M 4 = 64M 5 = 128M 6 = 256M 7 = 512M 8 = 1G 9 = 2G Others = Reserved
7:6	Win	Win Mode for data cache*
5	SingleUse	Reserved - set to '0' *
4	ExclusiveMode	EX-MODE
3	WrComb	WR-COMBINE*
2	Privilege	PRIV*
1	Writeable	WRITE*
0	Valid	1 register values valid 0 register values not valid (default)

* for MMU table region 0x05000000-0x05FFFFFF only

4.7.7. Tn local range root table 1 - MMCU_TnLOCAL_TABLE_PHYS1

Address: 0x04830708 + 0x20*n

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Tn Local Range Root Table Register1 (see also section 2.7)

Bit	Symbol	Description
31:2	Base	Physical Base 32-bit Word Address of MMU table first level for thread n
1:0	Reserved	

4.7.8. Tn global range root table 0 - MMCU_TnGLOBAL_TABLE_PHYS0

Address: 0x04830710 + 0x20*n

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Tn Global Range Root Table Register0 (see also section 2.7)

Bit	Symbol	Description
31:22	Base	Bit 31 to 22 of the Linear Base Address for thread n. The top bit must always be set to one.

Bit	Symbol	Description
21:12	Reserved	
11:8	Range	Range of Linear Address 0 = 4M 1 = 8M 2 = 16M 3 = 32M 4 = 64M 5 = 128M 6 = 256M 7 = 512M 8 = 1G 9 = 2G Others = Reserved
7:6	Win	Win Mode for data cache*
5	SingleUse	Reserved - set to '0' *
4	ExclusiveMode	EX-MODE
3	WrComb	WR-COMBINE*
2	Privilege	PRIV*
1	Writeable	WRITE*
0	Valid	1 register values valid 0 register values not valid(default)

* for MMU table region 0x05000000-0x05FFFFFF only

4.7.9. Tn global range root table 1 - MMCU_TnGLOBAL_TABLE_PHYS1

Address: 0x04830718 + 0x20*n

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Tn Global Range Root Table Register1 (see also section 2.7)

Bit	Symbol	Description
31:2	Base	Physical Base 32-bit Word Address of MMU table first level for thread n
1:0	Reserved	

4.7.10. Data Cache Enable - MMCU_DCACHE_CTRL

Address: 0x04830018

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:1	Reserved	

Bit	Symbol	Description
0	MMCU_DCACHE_CTRL	When set to 1, after the Dcache has been taken out of MMU bypass mode, this will enable data cache hits. The Dcache must first be flushed using the DCACHE_FLUSH register before enabling data cache hits.

4.7.11. Instruction Cache Enable - MMCU_ICACHE_CTRL

Address: 0x04830020
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:1	Reserved	
0	MMCU_ICACHE_CTRL	When set to 1, after the Icache has been taken out of MMU bypass mode, this will enable instruction cache hits. The Icache must first be flushed using the ICACHE_FLUSH register before enabling instruction cache hits.

4.7.12. Local region MMU enhanced bypass - MMCU_LOCAL_EBCTRL

Address: 0x04830600
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG
Local region MMU enhanced bypass mode control.

Bit	Symbol	Description
31:16	Reserved	
15:14	ICWin	Win Mode for instruction cache
13:8	Reserved	
7:6	DCWin	Win Mode for data cache
5:0	Reserved	

Note that the ICWin and DCWin controls are only applicable when cache lines are allocated(i.e. read miss).

4.7.13. Global region MMU enhanced bypass - MMCU_GLOBAL_EBCTRL

Address: 0x04830608
Reset Value: 0x00000000
Write Privilege: Controlled by TxPIOREG
Global region MMU enhanced bypass mode control.

Bit	Symbol	Description
31:16	Reserved	
15:14	ICWin	Win Mode for instruction cache
13:8	Reserved	

Bit	Symbol	Description
7:6	DCWin	Win Mode for data cache
5:0	Reserved	

Note that the ICWin and DCWin controls are only applicable when cache lines are allocated(i.e. read miss).

4.7.14. Enhanced bypass/wr combiner control - MMCU_TxEBWCCTRL

Address: 0x04830640 + 0x8*Thead_ID

Reset Value: 0x00000007

Write Privilege: Controlled by TxPIOREG

MMU enhanced bypass mode write combiner control.

Bit	Symbol	Description
31:4	Reserved	
7	Reserved	
6:4	Writeable	Control writeable flag for global/local regions in enhanced bypass mode 0 = writeable flag is 1 1 = Use byte address bit25, writeable flag is 1 when address bit is '1' 2 = Use byte address bit26, writeable flag is 1 when address bit is '1' 3 = Use byte address bit27, writeable flag is 1 when address bit is '1' 4 = Use byte address bit28, writeable flag is 1 when address bit is '1' 5 = Use byte address bit29, writeable flag is 1 when address bit is '1' 6 = Use byte address bit30, writeable flag is 1 when address bit is '1' 7 = writeable flag is 0 When a particular byte address bit is used to control writeable, that physical address bit would be set to '0'
3	Reserved	

Bit	Symbol	Description
2:0	WrComb	Control the write combine option for global/local regions in enhanced bypass mode 0 = Disable write combining for all addresses 1 = Use byte address bit25, write combine ON when address bit is '1' 2 = Use byte address bit26, write combine ON when address bit is '1' 3 = Use byte address bit27, write combine ON when address bit is '1' 4 = Use byte address bit28, write combine ON when address bit is '1' 5 = Use byte address bit29, write combine ON when address bit is '1' 6 = Use byte address bit30, write combine ON when address bit is '1' 7 = Enable write combining for all addresses. When a particular byte address bit is used to control write combine, that physical address bit would be set to '0'

4.7.15. Cache/MMU bypass control - SYSC_CACHE_MMU_CONFIG

Address: 04830028
Reset Value: 00000000
Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:7	Reserved	
6	DCACHE_HAVE_SKEW	Read only. Returns '1' if the Dcache supports skewed associativity
5	ICACHE_HAVE_SKEW	Read only. Returns '1' if the Icache supports skewed associativity
4:3	Reserved	
2:0	MMCU_DCACHE_CTRL	Set to 7 to put the MMU and caches into full TLB mode. Set to 6 to put both the caches into enhanced bypass mode. Set to 4 to put the instruction cache into enhanced bypass mode. Set to 2 to put the data cache into enhanced bypass mode. Set to 0 to put the caches into bypass mode.

4.7.16. JTAG debug control - SYSC_JTAG_THREAD

Address: 04830030
Reset Value: 0x00000000 or 0x00000004 depending on boot mode

Write Privilege: Controlled by TxPIOREG

This register determines the privilege level at which operations performed via the Meta core JTAG debug and slave interfaces operate.

If the core boots itself, the slave/debug privilege level is set to zero. Boot code can then hand privilege control over to the debug and slave ports by setting the bit to 1 when it is ready.

If the core is not configured to boot, the privilege bit is set automatically.

Bit	Symbol	Description
31:5	Reserved	
5:4	SLAVE_THREAD	Read only. These bits reflect the thread number currently selected for transactions that can be issued by the slave interface.
3	Reserved	
2	PRIVILEGE	1 - debug and slave interface is privileged 0 - debug and slave interface is unprivileged
1:0	DEBUG_THREAD	Read only. Thread number currently selected for transactions that can be issued by the debug port.

4.7.17. Data cache flush control - SYSC_DCACHE_FLUSH

Address: 04830038

Reset Value: 0x00000001

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:1	Reserved	
0	SYSC_DCACHE_FLUSH	When set to 1 this causes the data cache hardware to reset all cache line states to empty. The read state of this bit only changes to 1 when this operation is complete. This must only be used for initialisation of the caches following a reset.

4.7.18. Instruction cache flush control - SYSC_ICACHE_FLUSH

Address: 04830040

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

Bit	Symbol	Description
31:1	Reserved	
0	SYSC_ICACHE_FLUSH	When set to 1 this causes the instruction cache hardware to reset all cache line states to empty. The read state of this bit only changes to 1 when this operation is complete. This must only be used for initialisation of the caches following a reset.

4.7.19. Direct map addresses 0 - MMCU_DIRECTMAP0_ADDR

Address: 04830080

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

This read/write register allows the physical base addresses used for accesses in the Direct Mapped 0 part of the System Region to be defined. It provides the additional address bits required to complete the physical address generated.

Bit	Symbol	Description
31:23	MMCU_DIRECTMAP0_ADDR	Address base for 06000000-067FFFFF
22:16	Reserved	
15:14	ICWin	Win Mode for instruction cache
13:8	Reserved	
7:6	DCWin	Win Mode for data cache
5	SingleUse	Reserved - set to '0'
4	ExclusiveMode	EX-MODE
3	WrComb	WR-COMBINE
2	Privilege	PRIV
1	Writeable	WRITE
0	Mode	HTP/ATP Mode, 0 = ATP, 1 = HTP. When it is in ATP mode, only bits 31:23 are applicable

Note The ICWin and DCWin controls are only applicable when cache lines are allocated(i.e. read miss).

4.7.20. Direct map addresses 1 - MMCU_DIRECTMAP1_ADDR

Address: 04830090

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

This read/write register allows the physical base addresses used for accesses in the Direct Mapped 1 part of the System Region to be defined. It provides the additional address bits required to complete the physical address generated.

Bit	Symbol	Description
31:23	MMCU_DIRECTMAP1_ADDR	Address base for 06800000-06FFFFFF
22:16	Reserved	
15:14	ICWin	Win Mode for instruction cache
13:8	Reserved	
7:6	DCWin	Win Mode for data cache
5	SingleUse	Reserved - set to '0'

Bit	Symbol	Description
4	ExclusiveMode	EX-MODE
3	WrComb	WR-COMBINE
2	Privilege	PRIV
1	Writeable	WRITE
0	Mode	HTP/ATP Mode, 0 = ATP, 1 = HTP. When it is in ATP mode, only bits 31:23 are applicable

Note The ICWin and DCWin controls are only applicable when cache lines are allocated(i.e. read miss).

4.7.21. Direct map addresses 2 - MMCU_DIRECTMAP2_ADDR

Address: 048300A0

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

This read/write register allows the physical base addresses used for accesses in the Direct Mapped 2 part of the System Region to be defined. It provides the additional address bits required to complete the physical address generated.

Bit	Symbol	Description
31:23	MMCU_DIRECTMAP2_ADDR	Address base for 07000000-077FFFFF
22:16	Reserved	
15:14	ICWin	Win Mode for instruction cache
13:8	Reserved	
7:6	DCWin	Win Mode for data cache
5	SingleUse	Reserved - set to '0'
4	ExclusiveMode	EX-MODE
3	WrComb	WR-COMBINE
2	Privilege	PRIV
1	Writeable	WRITE
0	Mode	HTP/ATP Mode, 0 = ATP, 1 = HTP. When it is in ATP mode, only bits 31:23 are applicable

Note The ICWin and DCWin controls are only applicable when cache lines are allocated(i.e. read miss).

4.7.22. Direct map addresses 3 - MMCU_DIRECTMAP3_ADDR

Address: 048300B0

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

This read/write register allows the physical base addresses used for accesses in the Direct Mapped 3 part of the System Region to be defined. It provides the additional address bits required to complete the physical address generated.

Bit	Symbol	Description

Bit	Symbol	Description
31:23	MMCU_DIRECTMAP3_ADDR	Address base for 07800000-07FFFFFF
22:16	Reserved	
15:14	ICWin	Win Mode for instruction cache
13:8	Reserved	
7:6	DCWin	Win Mode for data cache
5	SingleUse	Reserved - set to '0'
4	ExclusiveMode	EX-MODE
3	WrComb	WR-COMBINE
2	Privilege	PRIV
1	Writeable	WRITE
0	Mode	HTP/ATP Mode, 0 = ATP, 1 = HTP. When it is in ATP mode, only bits 31:23 are applicable

Note The ICWin and DCWin controls are only applicable when cache lines are allocated(i.e. read miss).

4.7.23. Data cache partitioning thread 0 - SYSC_DCPART0

Address: 04830200

Reset Value: 0x00000000

Write Privilege: Controlled by TxPIOREG

This, and the following seven registers specify how the physical data cache and instruction cache resources are divided between threads. The global part of a thread's address map can be configured separately to the local part and, in theory, either part of any thread's cache resource may be shared with any other parts of any number of other threads. It is recommended however to use separate regions for each thread's local space and a common area for all threads' global spaces. Separate global spaces may be safely used for sub-sets of the threads provided the sub-sets are independent in terms of global memory usage.

Sizing is achieved by masking away address lines otherwise used to address 1/16th regions of the cache resource and adding in a fixed offset corresponding to the masked lines. It is invalid to attempt to operate the cache with fewer than four cache lines allocated to a region (unless it is never used), hence 1/16th partitioning of a 2k cache would not be permitted.

If the global caching regions for different threads are independent they can share read-only data or code in complete safety.

These registers cannot be configured in MMU bypass mode. They must be programmed initially after exiting MMU bypass mode and before enabling the caches for first use.

Bit	Symbol	Description
31	CachedWriteEnable	Cached write enable. When set, the MMU response for a write will be cached. This improves the performance for write intensive programs.
30:28	Reserved	

Bit	Symbol	Description
27:24	GlobalAddressOffsetT0	All values are potentially valid, for example: 0000 - first part of cache used 1000 - top half of cache used 0100 - second quarter of cache used etc…
23:20	Reserved	
19:16	LocalAddressOffsetT0	All values are potentially valid, for example: 0000 - first part of cache used 1000 - top half of cache used 0100 - second quarter of cache used etc…
15:12	PseudoCachePartition	Cache way partitioning mask for thread, Masks cache ways available to thread on cache miss. 0000 - No additional cache way masking 0001 - Way 0 masked … 1001 - Way 3 and Way 0 masked etc…
11:8	GlobalAddressMaskT0	Valid values are: 1111 - whole cache used 0111 - half of cache used 0011 - quarter of cache used 0001 - eighth of cache used 0000 - sixteenth of cache used
7:4	Reserved	
3:0	LocalAddressMaskT0	Valid values are: 1111 - whole cache used 0111 - half of cache used 0011 - quarter of cache used 0001 - eighth of cache used 0000 - sixteenth of cache used

4.7.24. Data cache partitioning thread 1 - SYSC_DCPART1

Address: 04830208

As above for thread 1.

4.7.25. Data cache partitioning thread 2 - SYSC_DCPART2

Address: 04830210

As above for thread 2.

4.7.26. Data cache partitioning thread 3 - SYSC_DCPART3

Address: 04830218

As above for thread 3.

4.7.27. Instruction cache partitioning thread 0 - SYSC_ICPART0

Address: 04830220

Bit	Symbol	Description
31:28	Reserved	
27:24	GlobalAddressOffsetT0	All values are potentially valid, for example: 0000 - first part of cache used 1000 - top half of cache used 0100 - second quarter of cache used etc…
23:20	Reserved	
19:16	LocalAddressOffsetT0	All values are potentially valid, for example: 0000 - first part of cache used 1000 - top half of cache used 0100 - second quarter of cache used etc…
15:12	Reserved	
11:8	GlobalAddressMaskT0	Valid values are: 1111 - whole cache used 0111 - half of cache used 0011 - quarter of cache used 0001 - eighth of cache used 0000 - sixteenth of cache used
7:4	Reserved	
3:0	LocalAddressMaskT0	Valid values are: 1111 - whole cache used 0111 - half of cache used 0011 - quarter of cache used 0001 - eighth of cache used 0000 - sixteenth of cache used

4.7.28. Instruction cache partitioning thread 1 - SYSC_ICPART1

Address: 04830228

As above for thread 1.

4.7.29. Instruction cache partitioning thread 2 - SYSC_ICPART2

Address: 04830230

As above for thread 2.

4.7.30. Instruction cache partitioning thread 3 - SYSC_ICPART3

Address: 04830238

As above for thread 3.