

Klessydra Processing Core Family

Technical Manual

(Preliminary edition – May 2020)



SAPIENZA
UNIVERSITÀ DI ROMA

Klessydra Processing Core Family

Technical Manual

Release Information

May.2020 preliminary edition

Copyright Notice

This book describes open hardware design.

Any part of the designs described in this book can be reproduced and used for any legal purpose.

Confidentially Notice

The information contained in this book is public.

Project Status

The information in this document is related to the design as of May. 2020.

Contents

Preface

About this book
Feedback

Chapter 1

Architecture overview

- 1.1 Features
- 1.2 Supported Instruction Set
- 1.3 Multi-threading model
- 1.4 Core Interfaces

Chapter 2

Memory model and protocol

- 2.1 Instruction Fetch
- 2.2 Memory Access Protocol
- 2.3 Misaligned Accesses
- 2.4 Memory Address Map

Chapter 3

Architecture registers

- 3.1 Register file
- 3.2 Control and Status Registers

Chapter 4

Pipeline Organization

- 4.1 General concepts
- 4.2 S0 core pipeline
- 4.3 T02x core pipeline
- 4.4 T03x core pipeline

Chapter 5

Exceptions and Interrupts

Chapter 6

Scratchpads (T1 only)

Chapter 7

DSP unit (T1 only)

Chapter 8

Debug Support

Chapter 9

Instruction Set

- 9.1 Integer Register-Immediate operations
- 9.2 Integer Register-Register Operations
- 9.3 Unconditional Jumps
- 9.4 Conditional Branches
- 9.5 Memory access Instructions
- 9.6 CSR Instructions

- 9.7 Timer and Performance Counter Instructions
- 9.8 Environment Call and Breakpoint instructions
- 9.9 Atomic Instructions
- 9.10 Klessydra Custom Extensions

List of Tables

Table.1.1	Clock, reset active low, test enable
Table.1.2	Initialization signals
Table 1.3	Program memory interface
Table 1.4	Data Memory interface
Table 1.5	Interrupt request / acknowledge
Table 1.6	Debug interface
Table 1.7	Miscellaneous control signals
Table 3.1	CSR Registers
Table.3.1.1	MSTATUS bits
Table.3.1.2	MCAUSE bits
Table.3.1.3	PCER bits
Table.3.1.4	MHPMCOUNTER bits
Table.3.1.5	MHPMEVENT bits
Table.3.1.6	MHARTID bits
Table.3.1.7	MIP bit
Table.5.1	Interrupt Handlers
Table 7.1	DSP instruction Listing
Table.8.1	Debug Registers
Table.8.2	Debug Control register bit map
Table.8.3	Debug Hit register bit map
Table.8.4	Debug Next Program Counter register bit map
Table.8.5	Debug Previous Program Counter register bit map
Table.9.1	Register-Immediate operations
Table.9.2	Register-Register Operations
Table.9.3	Unconditional Jumps
Table.9.4	Branches
Table.9.5	Load-Store Instructions
Table.9.6	CSR Instructions
Table.9.7	Privileged Instructions
Table.9.8	Atomic Instructions
Table 9.9	Klessydra Custom Extensions

List of Figures

Fig. 1.1 Conceptual view of hardware thread (hart) interleaved execution

Figure 2.1 Basic Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

Figure 2.2 Back to Back Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

Figure 2.3 Slow Response Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

Fig. 2.4 Klessydra Memory Map

Fig. 4.1 Generic pipeline microarchitecture scheme implemented in Klessydra cores.

Preface

About This Manual

This manual illustrates the architecture of the processors composing the Klessydra processing core family. As the Klessydra family includes several processors, each section of the manual first describes the common features of the cores and subsequently reports the differences among the cores, when applicable.

The intended audience of this manual is anyone who is interested in the Klessydra family as well as in RISC-V compliant microcontrollers, as a software programmer or a hardware designer.

Related documents

A.Traber, M. Gautschi, *Pulpino: Datasheet*, June 2017. Online.
<http://www.pulp-platform.org/documentation/>

A.Traber, M. Gautschi, D. Schiavone, *RI5CY: User Manual*, Nov. 2017. Online. <http://www.pulp-platform.org/documentation/>

D. Schiavone, *Zero-riscy: User Manual*, June 2017. Online.
<http://www.pulp-platform.org/documentation/>

A. Waterman, K. Asanovic, Editors, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, Version 2.2, May 2017, Online.
<https://riscv.org/specifications/>

A. Waterman, K. Asanovic, Editors, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, Version 1.10, May 2017, Online.
<https://riscv.org/specifications/>

Acknowledgement

This work was made possible by the technical support of the PULP research group.

Feedback

Address any feedback about the processing cores to the following addresses:

-mauro.olivieri@uniroma1.it, abdallah.cheikh@uniroma1.it,
mastrandrea@diet.uniroma1.it, menichelli@diet.uniroma1.it

Chapter 1

Architecture overview

1.1 Features

The Klessydra processing core family is a set of processors featuring full compliance with the RISC-V instruction set and intended to be placed within the Pulpino microprocessor platform. To date, the Klessydra family includes

- a minimal gate count single-thread core, **Klessydra S0**. The S0 core is not maintained as open-source;
- a class of multi-threaded cores, **Klessydra T0**, available in different implementations called Klessydra T0ab;
- a class of extended versions of the T0 cores, named **Klessydra T1** cores, featuring an MAU hardware accelerator.
- A class of fault tolerant versions of the T0 cores, featuring fault-tolerant mechanisms for harsh environment applications, named Klessydra F0x.

The Klessydra core family features:

- Full compliance with the RISC-V architecture specification (instruction set, control and status registers, interrupt handling mechanism and calling convention);
- Compliance with the standard RISC-V compilation toolchain;
- Interleaved multi-threaded execution of RISC-V *harts* (hardware threads);
- Easy and standardized multi-threading programming interface;
- Core synthesis on FPGA (presently, Xilinx Series 7 implementations have been tested);
- Hardware compliance with the Pulpino microprocessor platform, as pin-to-pin compatible alternative of the Pulpino RI5CY core;
- Software compliance with the Pulpino microprocessor platform, as compatible I/O memory map, interrupt handler memory map, program/data memory map;
- Extends the software test suite of Pulpino with custom tests designed specifically for the klessydra cores.

1.2 Naming convention

The different cores available in the Klessydra family follow the naming convention depicted in Fig. 1.1.

1.3 Supported Instruction Set

To date, all the Klessydra cores implement the 32-bit integer RISC-V machine mode instruction set, namely user-level RV32I base integer instruction set version 2.1 and M-mode privileged instruction set version 1.1. T0 and T1 cores support the RV32IME set.

The T0 and T1 cores support the atomic instruction AMOSWAP.W from the RVA atomic instruction extension.

The T1 core extends the instruction set with non-native **custom** vector instructions for memory to scratchpad transfers and vector arithmetic operations. Vector instructions come in three different variants supporting different data width “8-bit, 16-bit, 32-bits” e.g.

Only M-mode operation is supported, so that no operating system support is implemented. Yet, the Klessydra family comes with a baseline runtime system software layer that implements part of the interrupt handling features and part of the multi-threaded programming model.

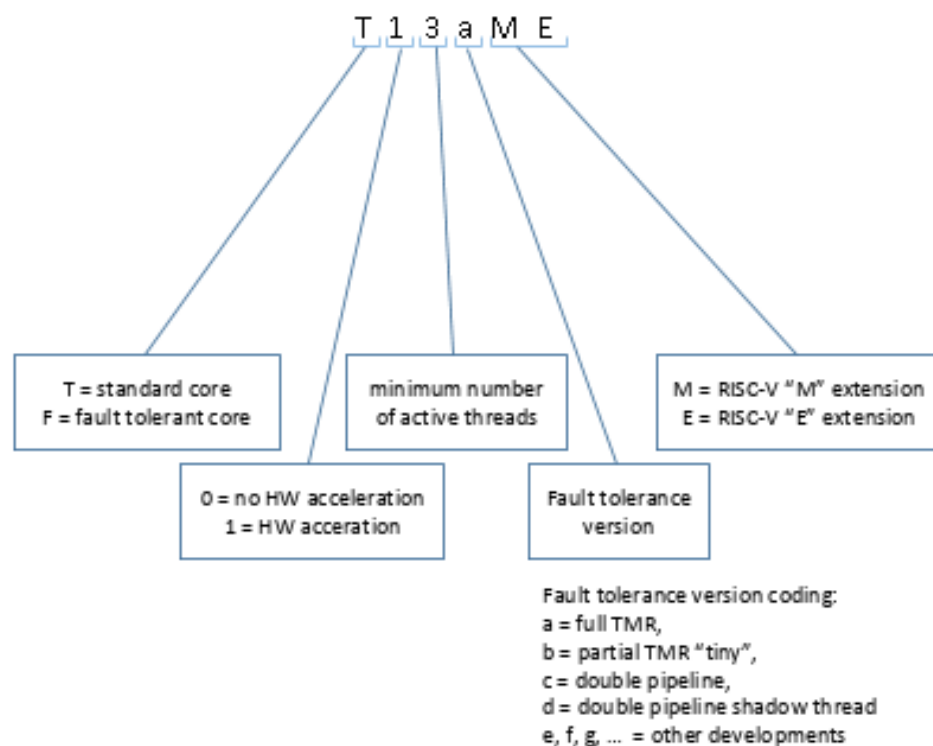


Fig. 1.1. Naming convention

1.4 Multi-threading model

Klessydra S0 core supports single thread execution (RISC-V hart) only, with the following features:

- The hart can be interrupted by a trap such as an external interrupt or instruction exception. Software interrupts are supported, although their use is expected to be impractical in a single-thread execution environment. When the trap handling routine ends the core resumes the original execution thread (see Chapter “Exception and Interrupts” for details);
- The core can enter an idle state by means of the WFI instruction; when an external interrupt arrives at the core, the core starts the execution of the interrupt handling routine as the new hart of execution.

- The hart can be halted and resumed by means of the *Fetch_en* core interface signal.

Klessydra T0x, T1x and F0x cores implement interleaved multi-threading. At each clock cycle, a new instruction is fetched from a different hart (Fig. 1.2).

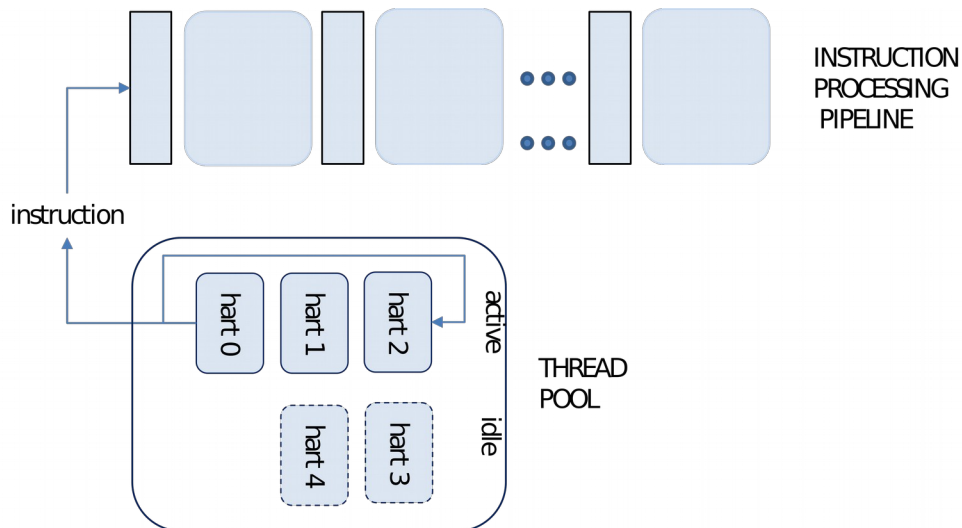


Fig. 1.2. Conceptual view of hardware thread (hart) interleaved execution

The execution has the following features:

- An idle hart can be activated by an interrupt request directed to the hart. The core executes the interrupt handling routine within the hart. When the interrupt handling routine ends, the hart becomes idle again (see Chapter “Exception and Interrupts” for details).
- An active hart can be interrupted by instruction exceptions or interrupt requests. When the interrupt/exception handling routine ends and the signal **fetch_enable_i** is high, the core resumes the interrupted execution hart. (see Chapter “Exception and Interrupts” for details);
- An active hart can become idle by executing the WFI instruction;
- The maximum number of active harts is an architecture characteristic parameter called Thread Pool Size.
- Each hart is identified by an integer number ranging from 0 up to Thread Pool Size - 1.
- There is also a minimum number of active harts, needed to avoid data hazards between threads during the pipelined execution, called Thread Pool Baseline. The Thread Pool Baseline value is an architecture characteristic parameter related to the instruction pipeline organization implemented in the hardware microarchitecture of the core.
- When the number of active threads is less than the Thread Pool Baseline, one or more idle hart runs in the pipeline as NOP instructions.

As a general note, a higher Thread Pool Baseline value corresponds to a higher sustainable clock frequency and generally indicates a higher

performance when running at full thread pool. For example, a T03 core will significantly outperform a T02 core when executing 4 harts.

1.5 Core Interfaces

The core interface is signal-to-signal compatible with the Pulpino microprocessor platform, and as such it is the same as Pulpino RI5CY cores. The detailed description follows.

Table.1.1 Clock, reset active low, test enable

Name	Direction	Width	Notes
clk_i	In	1	Core clock signal
clock_en_i	In	1	Core clock enable
rst_ni	In	1	Core reset signal, active low
test_en_i	In	1	Core test enable (unused)

Table.1.2 Initialization signals

Name	Direction	Width	Notes
boot_addr_i	In	32	Boot address value
core_id_i	In	4	Core id number
cluster_id_i	In	6	Cluster id number

Table 1.3 Program memory interface

Name	Direction	Width	Notes
instr_req_o	Out	1	Request signal, must stay high until accepted
instr_gnt_i	In	1	Request accepted, address may change in the next cycle
instr_rvalid_i	In	1	Instruction valid, stays high for exactly one cycle.
instr_addr_o	Out	32	Address
instr_rdata_i	In	32	Instruction read from memory

Table 1.4 Data Memory interface

Name	Direction	Width	Notes
data_req_o	Out	1	Request signal, must stay high until accepted
data_gnt_i	In	1	Request accepted, address may change in the next cycle
data_rvalid_i	In	1	Data valid, stays high for exactly one cycle
data_we_o	Out	1	Write enable, high = write, low = read
data_be_o	Out	4	Byte selection

data_addr_o	Out	32	Address
data_wdata_o	Out	32	Data to be written to memory
data_rdata_i	In	32	Data read from memory
data_err_i	In	1	Memory error signal

Table 1.5 Interrupt request / acknowledge

Name	Direction	Width	Notes
irq_i	In	1	Interrupt request signal
irq_id_i	in	5	Interrupt request vector value
irq_ack_o	out	1	Interrupt acknowledge signal
irq_id_o	in	5	Interrupt acknowledge vector value (unused)

Table 1.6 Debug interface

Name	Direction	Width	Notes
debug_req_i	In	1	Debug request
debug_gnt_o	Out	1	Debug request granted
debug_rvalid_o	Out	1	Debug data valid
debug_addr_i	In	15	Debug location address
debug_we_i	In	1	Debug write enable
debug_wdata_i	In	32	Debug data to be written to core
debug_rdata_o	Out	32	Debug data read from core
debug_halted_o	Out	1	Debug halt acknowledge
debug_halt_i	In	1	Debug halt request
debug_resume_i	in	1	Debug resume signal

Table 1.7 Miscellaneous control signals

Name	Direction	Width	Notes
fetch_enable_i	In	1	Fetch enable, stops the core
core_busy_o	Out	1	Core busy signal
ext_perf_counters_i	In	1	External performance counter signal (unused)

Chapter 2

Memory model and protocol

2.1 Instruction Fetch

The instruction fetch stage of the core is called FSM_IF and is able to supply one instruction to the instruction decode stage per cycle, if the program memory is able to serve one instruction per cycle. Instructions are word aligned, meaning that the two least significant bits in the PC are always set to 0, and the PC value is incremented by 4 units at each new fetch when no branch occurs. Compressed instruction format is not supported. No prefetch logic is present.

2.2 Memory Access Protocol

The program and data memory access protocol is pin-to-pin compatible with the Pulpino microprocessor platform, and as such it is the same as RI5CY / Zeroriscy cores'. The protocol that used to access the data memory works as follows. The program memory follows the same protocol except for the absence of write operation support.

The core provides a valid address in *data_addr_o* and sets *data_req_o* high. The memory then answers with *data_gnt_i* set high as soon as it is ready to serve the request. This may happen in the same cycle as the request is sent or any number of cycles later. After a grant is received, the address may be changed in the next cycle by the core. In addition, the *data_wdata_o*, *data_we_o* and *data_be_o* signals may be changed. After receiving a grant, the memory answers with *data_rvalid_i* set high if *data_rdata_i* is valid. This may happen one or more cycles after the grant has been received. The signal *data_rvalid_i* must also be set when a write operation is performed, although the *data_rdata_i* has no meaning in this case. Figure 2.1, Figure 2.2 and Figure 2.3 shows examples of the protocol timing.

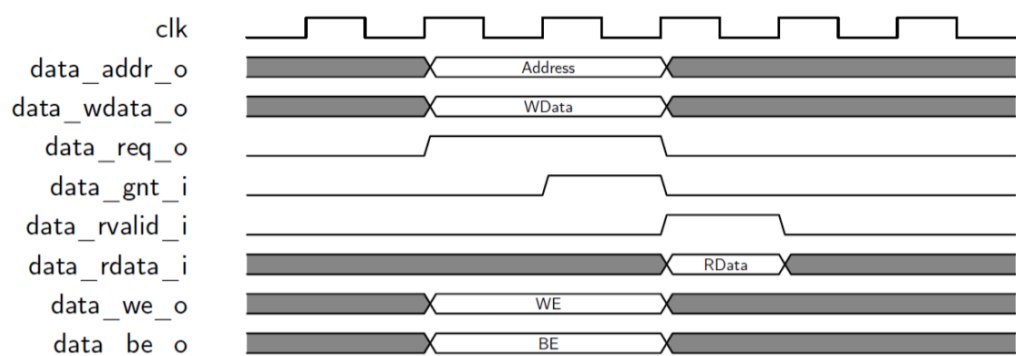


Figure 2.1 Basic Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

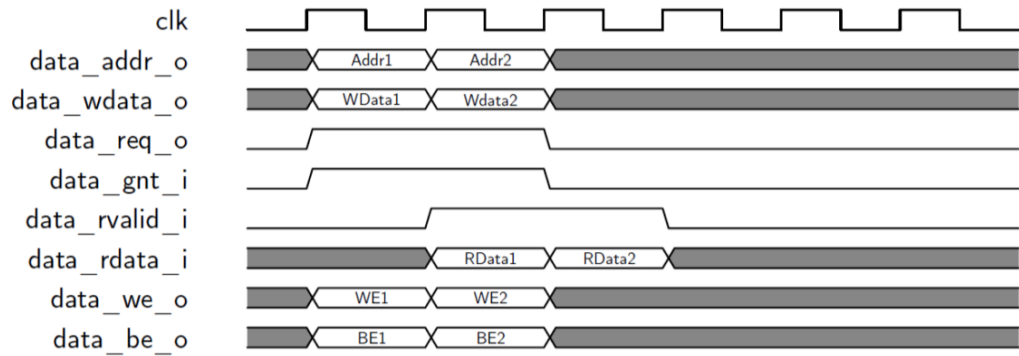


Figure 2.2 Back-to-Back Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)



Figure 2.3 Slow Response Memory Transaction (reprinted from RI5CY manual, rel. Jan 2017)

2.3 Misaligned Accesses

The core hardware does not perform misaligned accesses natively (i.e. accesses that are not aligned on natural word boundaries). If a misaligned memory access is requested by an instruction, the core produces an exception. There is no necessary hardware to realize the misaligned access by multiple aligned access. In compliance with RISC-V specification, misaligned accesses are therefore not guaranteed to be atomic.

2.4 Memory Address Map

Harts (i.e. hardware threads) running on a Klessydra core share the memory map illustrated in Fig. 2.4, which is compliant with the Pulpino SoC platform specification.

The MIP CSR, one for each hart, are memory mapped starting at address 0x0000ff00 and allow for inter-thread interrupts, in compliance with the RISC-V specification. (Other CSRs are not memory mapped).

Each hart has its own stack, and the stack size and starting address are customizable at software level in the runtime system startup routine. The remaining memory space is available for inter-thread data communication.

For information about the addresses from 0x00 to 0x90, see the vector table in chapter 5. Address 0x94 is reserved to MTVEC.

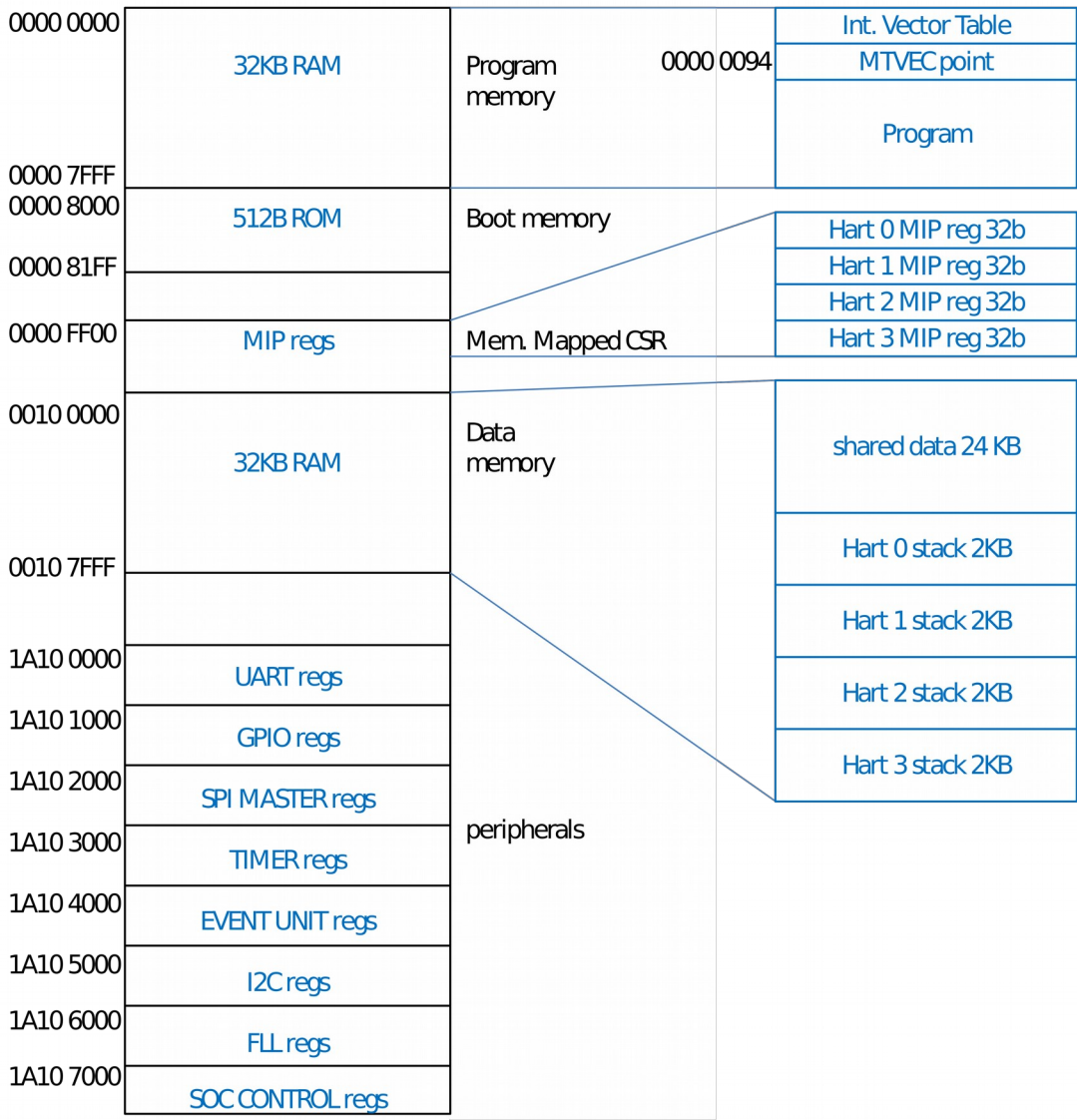


Fig. 2.4 Klessydra Memory Map (assuming 4 Threads, 2 KB stack per thread)

Chapter 3

Architecture Registers

3.1 Register File

Klessydra has 32x32-bit wide registers which form the registers x0 to x31. Register x0 is statically bound to 0 and can only be read. Write on register x0 has no side effect. They can be modified to 16x32 registers if the RV32E embedded extension was enabled.

3.2 Control and Status Registers

Klessydra cores implement a subset of the control and status registers specified in the RISC-V privileged specification, limited to the registers needed for M-mode operation and to the functionalities implemented in the core. Klessydra cores also implement some additional CSRs specifically needed for the core operations and/or for compliance with the Pulpino microprocessor platform. This extended CSR sub-set is composed of the MIRQ, PCER, PCMR registers. The whole set of CSRs implemented in the Klessydra cores is as follows:

Table 3.1 CSR Registers

Name	CSR Address	Reset Value	R/W	Description
MSTATUS	0x300	0x0000_1808	R/W	Machine Status
MEPC	0x341	0x0000_0000	R/W	Machine Exception Program Counter
MCAUSE	0x342	0x0000_0000	R/W	Machine Trap Cause
PCER	0x7A0	0xFFFF_FF	R/W	Performance Counter Enable
MHPMCOUNTER	0xB00,0xB02 0xB03, 0xB06-0xB0A	0x0000_0000	R/W	Machine Performance-Monitoring Counter
MHPMEVENT	0x323, 0x326-0x32A	0x0000_0000	R/W	Machine Performance-Monitoring Event Selector
MCPUID	0xF00	0x0000_0100	R	CPU Description
MIMPID	0xF01	0x0000_8000	R	Implementation ID
MHARTID	0xF10	-	R	Hardware Thread ID

MIP	0x344	-	R/W	Interrupt Pending
MTVEC	0x305	0x0000_0094	R/W	Trap-Handler Base Address
MBADADDR	0x343	0x0000_0000	R/W	Misaligned Address Container
MIRQ	0xFC0	-	R	Interrupt Request
MVSIZE	0xBF0	0x0000_0001	R/W	Set Vector Size unit (T1)
MVTYPE	0xBF8	0x0000_0002	R/W	Set the data type (T1)
MPSCLFAC	0xBE0	0x0000_0000	R/W	Set the post scaling factor (T1)

- **MSTATUS Register bit map**

Table.3.1.1 MSTATUS bits

Bit #	R/W	Description
3	R/W	<p>Interrupt Enable: When an exception is encountered, Interrupt Enable will be set to 1'b0, and it's state will be stored in bit '7'.</p> <p>When the <i>mret</i> instruction is executed, the original value of Interrupt Enable will be restored from the 7th bit.</p> <p>If you want to enable interrupt handling in your exception handler, set the Interrupt Enable to '1' inside your handler code.</p>
7	R/W	<p>Interrupt Previous Enable: Takes the state of the 3rd bit when serving an interrupt, and when an <i>mret</i> is served it stays latched to 1. And returns the 3rd bit back to it's original value.</p>

- **MEPC Register**

When an exception is encountered, the current program counter is saved in MEPC, and the core jumps to the MTVEC address. When an MRET instruction is executed, the value from MEPC replaces the current program counter, unless the return value was a WFI instruction, in this case we return to the instruction in the address after the WFI.

- **MCAUSE Register bit map**

Table.3.1.2 MCAUSE bits

Bit #	R/W	Description
31	R	Interrupt: This bit is set when the exception was triggered by an interrupt.
30	R	WFI: This bit indicates that the last instruction before entering the subroutine was a <i>WFI</i>
4:0	R	Trap Cause: "0011" for SW IRQ, "0111" for Timer IRQ, "1011" for

- **PCER Register bit map**

Each bit in the PCER register controls one performance counter. If the bit is 1, the counter is enabled and starts counting events. If it is 0, the counter is disabled and its value won't change.

Table.3.1.3 PCER bits

Bit #	Description
9	Branch Taken Counter Enable
8	Branch Counter Enable
7	Jump Counter Enable
6	Store Counter Enable
5	Load Access Counter Enable
4	Instruction Miss Counter Enable (currently not implemented)
3	Jump Access Stall Counter Enable (currently not implemented)
2	Load/Store Access Stall Counter Enable
1	Instruction Counter Enable
0	Cycle Counter Enable

- **MHPMCOUNTER Registers**

Klessydra Core includes a MCYCLE counter, a MINSTRET counter and others 6 additional event counters, MHPMCOUNTER3, MHPMCOUNTER6-MHPMCOUNTER10 of which only the first eight are used. The names of the registers are compliant to RISC-V but the counters are not divided into 32 lower bits and 32 higher bits. Only MCYCLE and MINSTRET are extended to 64 bits by the registers CYCLEH and MINSTRETH. The counter value is 32 bits unsigned integer.

Table.3.1.4 MHPMCOUNTER bits

Register	Description
MCYCLE	Counts the number of cycles the core was active (not sleeping)
MINSTRET	Counts the number of instructions executed
MHPMCOUNTER3	Number of load/store data hazards
MHPMCOUNTER4	currently not used
MHPMCOUNTER5	currently not used
MHPMCOUNTER6	Number of data memory loads executed
MHPMCOUNTER7	Number of data memory stores executed

MHPMCOUNTER8	Number of unconditional jumps
MHPMCOUNTER9	Number of branches. Counts taken and not taken branches
MHPMCOUNTER10	Number of taken branches

- **MHPMEVENT Registers**

In each MHPMEVENT register all the bits are statically bound to 0 except for the bit related to the counter that must be enabled. If that bit is 1, the counter is active and starts counting events. For instance, if the user wants to enable MHPMCOUNTER3 he will set the bit #2 (the 3th bit) of MHPMEVENT3 to 1. This procedure is equivalent to set PCER (3) to 1. The core includes 6 registers, MHPMEVENT3, MHPMEVENT6-MHPMEVENT10.

Table.3.1.5 MHPMEVENT bits

Register	Not Bound Bit #
MHPMEVENT3	2
MHPMEVENT4 (currently not used)	-
MHPMEVENT5 (currently not used)	-
MHPMEVENT6	5
MHPMEVENT7	6
MHPMEVENT8	7
MHPMEVENT9	8
MHPMEVENT10	9

- **MCPUID Register**

The value of this register is fixed to 256 and cannot be changed. By using the CPUID opcode, software can determinate processor type and the presence of features.

- **MIMPID Register**

The value of this register is fixed to 32768 and cannot be changed. MIMPID provides a unique encoding of the version of the processor implementation.

- **MHARTID Register**

This register contains the integer ID of the hardware thread running the code. His value depends on Cluster and Core external signals and can only be read.

Table.3.1.6 MHARTID bits

Bit #	Description
9:4	ID of the Cluster

3:0 ID of the core within the cluster

- **MIP Register**

The MIP register contains information about the type of pending interrupts. Bits #11 and #7 are enabled according to the external interrupt bits while bit #3 is settled to 1 to activate the SW interrupt routine.

Table.3.1.7 MIP bits

Bit #	R/W	Interrupt Type
11	R	External Interrupt
7	R	Time Interrupt
3	R/W	Software Interrupt

- **MTVEC Register**

When an exception or an interrupt occurs, PC is loaded with the value of this register. MTVEC is the standard RISC-V base trap vector.

- **MIRQ Register**

This register saves which interrupt has been called. The value of this register is four times the number of the interrupt's bit enabled. For instance, if `irq_i(3)` is set, MIRQ will be loaded with 12. If no interrupt is set, MIRQ value is 65535, that is just an arbitrary number.

- **BADADDR Register**

When an instruction-fetch, load or store address-misaligned or access exception occurs, MBADADDR is written with the faulting address.

- **MVSIZE Register**

Setting this register will set the vector size to be used by the mathematical unit. The biggest size should not exceed the SPM size, since overflow bits will be ignored.

- **MPSCLFAC Register**

Contains the post scaling factor that determines the shift amount in KDOTPPS custom Klessydra instruction.

Chapter 4

Pipeline Organization

4.1 General concepts

Klessydra cores implement pipelined instruction processing. The number of pipeline stages differs among the cores as reported below. In the following, **F** indicates instruction fetch, **D** indicates operand read from register file and instruction decoding, **E** indicates operation execution, **W** indicates result writeback to the register file.

In all cores, the **F** stage latency is equal to the latency of program memory access, and variable latency program memory is supported (as for the case of instruction cache memory). The **F** stage latency is 1 in case of single-cycle-access program memory.

For other pipeline stages, the latency may be fixed or depend on external events (e.g. data memory latency, contention on CSR updating in case of interrupt requests). When a stage latency takes more than 1 cycle, the hardware stalls the preceding stage by local handshake signals. Similarly, each stage locally signals the succeeding stage when a new item is ready.

The generic microarchitecture for T0 cores is depicted in Fig. 4.1. Each-thread is identified by a positive integer number *harc* (hardware context). The *harc* counter changes the *harc* value at each new instruction fetch, and the *harc* value associated to an instruction is passed through the pipeline stages. Most of the logic in the pipeline control section is replicated on a per-thread basis, and the *harc* value is used to properly index the logic units. Conversely, all the logic in the processing pipeline is not per-thread replicated with the only exception of the data register file. In the S0 core, per-thread replication and the *harc*-related logic are natively absent.

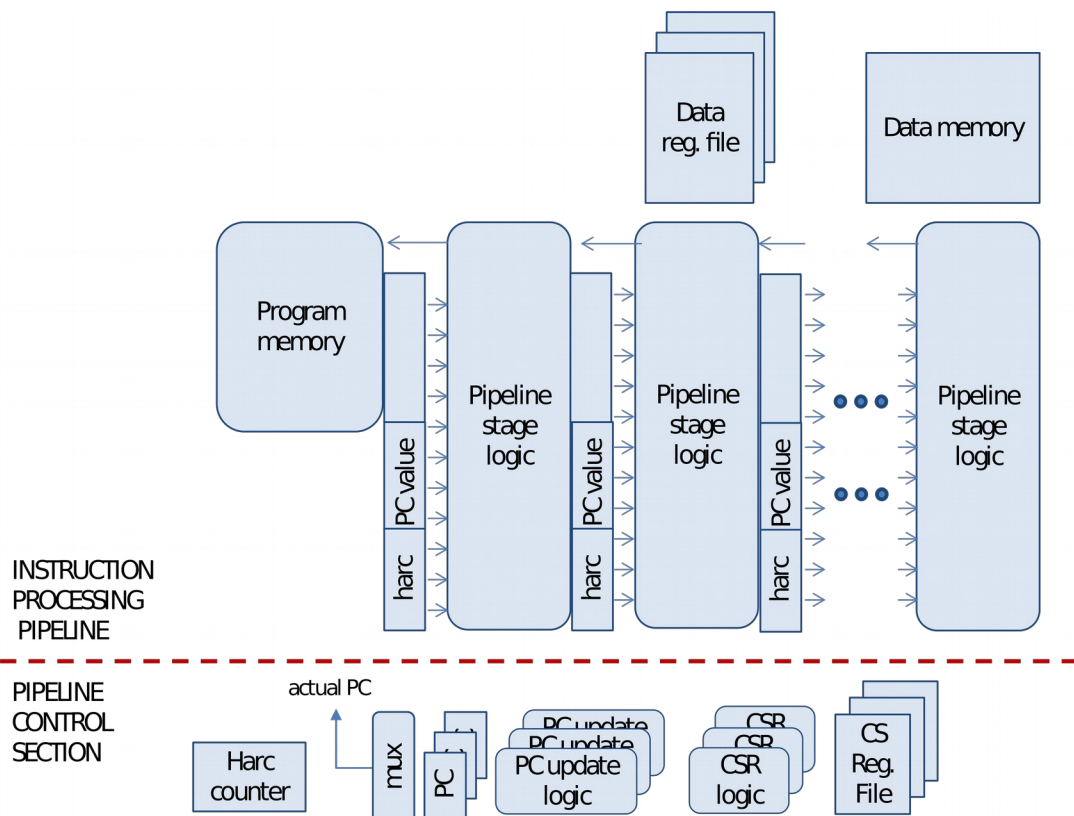


Fig. 4.1 – Generic pipeline microarchitecture scheme implemented in Klessydra T0 cores.
 The specialize microarchitecture of T1 cores is represented in Fig 4.2.

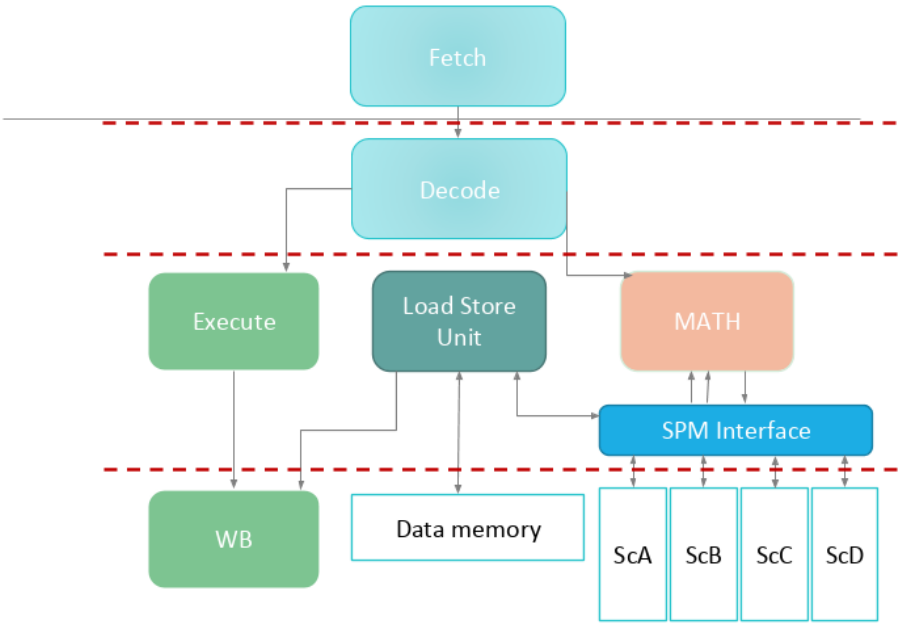


Fig. 4.2 – Datapath sketch of T1 cores.

T1 cores feature an execution stage that is split into a mathematical acceleration unit, scratchpad memory unit and a regular execution unit.

4.2 S0 core pipeline

The Klessydra S0 core implements a 2-stage pipeline according to the model **F / DEW**. The latency scheme is as follows:

	F	DEW
Load and store instructions	≥ 1	≥ 2
CSR instructions	≥ 1	≥ 2
All other instructions	≥ 1	1

Branch instructions are predicted as not-taken and are executed with a delay slot of 1 cycle; in case of taken branch the hardware flushes any wrongly fetched instruction from the pipeline.

Data hazards never occur.

4.3 T02x core pipeline

The Klessydra T0x cores implement a 3-stage pipeline according to the model **F / D / EW**. The latency scheme is as follows:

F	D	EW
----------	----------	-----------

Load and store instructions	≥ 1	1	≥ 2
CSR instructions	≥ 1	1	≥ 2
Atomic memory operations	≥ 1	1	≥ 4
All other instructions	≥ 1	1	1

Branch instructions are predicted as not-taken and are executed with a delay slot of 2 cycles; in case of taken branch the hardware flushes any wrongly fetched instruction, belonging to the branching thread, from the pipeline. No pipeline flush occurs if at least 3 threads are interleaved in the pipeline.

Data hazards never occur, provided that at least 2 threads (Thread Pool Baseline) are interleaved in the pipeline.

4.4 T03x / T13x / Fxxx core pipeline

The Klessydra T03x/T13x cores implement a 4-stage pipeline according to the model F / D / E / W. The latency scheme is as follows:

	F	D	E	W
Load and store instructions	≥ 1	1	≥ 2	0
CSR instructions	≥ 1	1	≥ 2	0
Atomic memory operations	≥ 1	1	≥ 4	0
All other instructions	≥ 1	1	1	1
Specialized vector instructions	≥ 1	1	≥ 2	0

Branch instructions are predicted as not-taken and are executed with a delay slot of 3 cycles; in case of taken branch the hardware flushes any wrongly fetched instruction, belonging to the branching thread, from the pipeline. No pipeline flush occurs if at least 3 threads are interleaved in the pipeline.

Data hazards never occur, provided that at least 2 threads (Thread Pool Baseline) are interleaved in the pipeline.

Chapter 5

Exceptions and Interrupts

Klessydra cores implement exceptions on illegal instructions, on load and store instructions to invalid addresses, on misaligned memory accesses, and on ECALL instruction execution.

Klessydra cores implement vectorized interrupts, specifically supporting 32 separate interrupt service routines. There are three types of interrupt:

- Software Interrupt
- External Interrupt
- Timer Interrupt

The interrupt/exception vector table supported by Klessydra cores is compliant with the Pulpino platform interrupt vector table, as follows:

Table.5.1 Interrupt Handler address map

0x00-0x7C	Interrupts 0-31
0x80	Reset
0x84	Illegal Instruction
0x88	ECALL Instruction Executed
0x8C	LSU Error (Invalid Memory Access)
0x90	Software Interrupt

Except Code	Exception
0x0000_0002	ILLEGAL_INSN_EXCEPT_CODE
0x0000_0005	LOAD_ERROR_EXCEPT_CODE
0x0000_0007	STORE_ERROR_EXCEPT_CODE
0x0000_000B	ECALL ECALL_EXCEPT_CODE
0x0000_0004	LOAD_MISALIGNED_EXCEPT_CODE
0x0000_0006	STORE_MISALIGNED_EXCEPT_CODE
0x0000_0100	ILLEGAL_VECTOR_SIZE_EXCEPT_CODE
0x0000_0101	ILLEGAL_ADDRESS_EXCEPT_CODE
0x0000_0102	SCRATCHPAD_OVERFLOW_EXCEPT_CODE

Interrupt handling is accomplished in the core hardware by jumping to the address contained in MTVEC, in compliance with RISC-V specification; the pre-compiled startup software routine located at MTVEC address implements the interrupt vector table as it is shown above, jumping to the right handler routine address. The interrupt handler are to be written by the final user according to the target application.

Interrupts can be enabled/disabled on a global basis through the MSTATUS register; they cannot be individually enabled/disabled. Exceptions cannot be disabled.

When entering an interrupt routine, the core saves the current value of MIE (3rd-bit) to the MPIE (7th-bit) in the MSTATUS register; the state of MIE will be restored after returning from interrupt service routine.

If multiple interrupt requests arrive at the same cycle, the order of service is external interrupt first, then software interrupt, timer interrupt and exceptions (compliance to RISC-V specification).

In T0 cores, external interrupts are always re-directed to hart number 0. Software interrupts can be directed from any active hart, to any active or idle hart. Software interrupts allow inter-hart service requests.

In T0 cores and in T1 cores, as all status registers are replicated on a per-thread basis, the interrupt/exception handling mechanism is implemented referring to the status registers of the interrupted thread.

T1 cores introduce five more exceptions regarding the scratchpad handling. Exceptions will be raised if the Math Accelerator unit operands are from non-scratchpad addresses, or if writing or reading will result in a request from an overflowed scratchpad address, or if we have dual writes or dual reads from the same scratchpad such as in the case of the LSU and Math Accelerator unit working simultaneously

Chapter 6

Scratchpads and mathematical unit (T1 version only)

6.1 Scratchpad memory subsystem

Klessydra T1 cores include scratchpad memories, with configurable number of scratchpads, banks, and scratchpad size and address mapping. The configurations can be modified in the PKG file of the synthesizable Klessydra suite. Each scratchpad memory (SPM) is composed of a set of memory banks; the number of banks available in each SPM is defined by on the “SIMD” parameter value set in the PKG file.

Each bank address holds a 32-bit word. An SPM data line is composed of as many words as the “SIMD” value. As addresses remain byte-aligned, the address distance between SPM data lines is $+ \text{SIMD} * 4$. Each word on the line has its own address and can be independently accessed (with 4-byte aligned address).

Any SPM bank can be read or written to. For read access, any bank that is not bank0 will cause the data read in SIMD fashion to be rotated as if it was coming from bank0 by a read rotator. While for write access, any write to a bank different from bank0 will cause the data to be rotated to its correct destination bank by a write rotator. The rotators were made to align the two input source operands

Each scratchpad has one read port, and one write port. Each port has size $\text{SIMD} * 32$ -bits (e.g. for $\text{SIMD}=4$, we have $4 * 32 = 128$ bits).

The SPMs can be accessed by the MAU or the LSU. When a dual read (or dual write) access is requested to the same SPM on the same port by two different units (MAU and LSU), priority will be given to the unit that requested the access first and the other unit will be halted until the operation is finished. Due to the in-order single-issue pipeline of the Klessydra cores it is not possible that the two units request access to the SPM in the same cycle).

All transfers to/from the scratchpads go through an interface called **SPMI**. Both SPM read and writes happen through this SPMI wrapper. The LSU and MAU are the only units that interact with the scratchpad memories, always through the SCI.

6.2 Mathematical accelerator unit

The Mathematical unit was designed to execute custom Klessydra instructions targeting vector, DSP-like and CNN-inference-like operations.

The Mathematical unit interfaces to the SPMI unit by means of two read ports for the operands coming from the scratchpad memory interface, and one write port to the scratchpad memory interface. The read and write port width are dependent on the SIMD parameter value set in the PKG file.

The custom instruction set executed in the Mathematical unit are listed in table 7.1. It executes different instructions many of which have different variants. Table.7.1 also shows the SIMD capability of the Mathematical unit. A composition of partial functional units has

been adopted to enhance the SIMD execution and to optimize the area consumption mathematical unit. Addition instructions use a combination of 8-bit adders to make 8-bit, 16-bit, and 32-bit additions. Multiplication instructions use a combination of 16-bit multipliers to perform 8-bit, 16-bit and 32-bit multiplications¹. There are no 32-bit arithmetic units in the mathematical except for the 32-bit shifters, that can be configured to do 8,16,32-bit right arithmetic or logical shifts, accumulators (32-bit adders, and 16-bit adders for both 8, and 16-bit), and Rectify Linear Unit (RELU).

When working on vectors, the Mathematical unit exploits built-in hardware loop (zero overhead loops), executing the following steps in hardware:

- a. Increment the source and destination vector pointers to fetch the next element chunk;
- b. Decrement the remaining number of elements to process;
- c. Evaluate a conditional branch to check whether the number of remaining elements reached zero.

The Mathematical unit can operate in parallel with respect to the other execution units. Since the custom Klessydra instructions never have dependencies with the standard RISC-V instructions, the IE unit and LSU can work in parallel with the Mathematical unit.

The Mathematical unit recovers its state when a halt occurs due to dual read/write access.

¹ 16-bit multipliers were chosen over 8-bit multipliers since doing 32-bit multiplication using 8-bit multipliers would be inefficient, and also 16-bit are the optimal choice needed for utilizing DSP blocks on presently available FPGAs

Chapter 7

Fault Tolerance Support (F0x versions only)

Klessydra core versions F0x support several mechanisms of fault tolerance targeting aerospace and safety critical application. Most of the mechanisms implemented address tolerance to single event upset (SEU) in memory elements (registers and memories).

7.1 Basic mechanisms

Supported standard FT mechanisms are Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR), both based on repetition on functional modules and comparison of outputs through a voting system.

DMR uses two replicas of combinational or sequential logic, it can only detect errors and has a low area occupation and power consumption but high time of implementation.

Basic-TMR is a triple repetition of combinational or sequential logic and a majority voter; it has the same time of implementation of DMR and also area and power consumption increase.

Full-TMR adds a triple redundancy to both logic and registers at cost of area and power consumption and it uses cross voters to guarantee high error correction capability.

Global-TMR is based on full-TMR but it can be automated through synthesis tools.

7.2 F03a: Fully TMR - Partial TMR Design

The protection of control and state registers is a priority because they contain vital information about core operation and they are written only once at first run, so a TMR must be used.

Counter registers are less critical because they are constantly refreshed. Each core has a dedicated counter with many 32-bit registers, so it's suggested to use alternative techniques:

- MSB-TMR: triple redundancy only of N most significant bits, reducing area impact.
- DMR: detection of an error trigger a trap identified by a code and it's managed by the software.
- Software protection: no hardware protection, the software periodically reads and compares counters.

Pipeline robustness is fundamental in TMR because redundancy does not protect registers from loading wrong values that irredeemably corrupt code execution. So redundancy has to be applied to all registers between pipeline stages and state registers of state machines.

Registers file are dedicated for each thread so a TMR has the highest impact in terms of area. The voting system also lengthens critical path that lowers the maximum clock frequency.

Program counter unit has a dedicated 32-bit register for each thread and some flip-flop to store events and conditions that must be resolved from the unit. Flip-Flop corruption doesn't lead to loss of control because PC update is defined by signals coming from pipeline. An error on exception service Flip-Flop is more critical because it requires a response from CSR. Due to the small area occupation on few registers and Flip-Flop it's suggested to protect with TMR the PC unit.

7.3 F03b: Double Pipeline Design with Check&Restore

In this version, a protection technique is used that does not allow error correction, but the area occupied by the TMR version is reduced by a third, without losing reliability. This design implies a change of the internal architecture of the core. The structure is based on a new Processing Unit composed of two Pipelines, the CRU and a new CSR unit derived from the TMR version. In this architecture, a checkpoint is created before critical portions of code or periodically. Checkpoint control is managed by the Check Restore Unit (CRU) and the CSR.

The two pipelines are the same as the Klessydra T03 version. The input and output signals of the pipelines are controlled by the CRU, which can drive them exclusively to start checkpoint or restore procedures, not natively implemented in the Pipeline. To allow CRU functioning and error check, there is an internal register, called CRSTATUS. This critical register is protected by TMR redundancy and can be read (but not written via) CSR instructions. The management of the check and restore system requires instructions for management and control. These instructions are:

- Chepoint start instruction
- Instruction to activate thread dependency
- Instruction to restart the restore manually
- Instruction to deactivate protected mode

The double pipeline structure adds two operating modes to the system:

- the normal mode - it allows to deactivate the clock of the not used pipeline in order to reduce the dynamic consumption of the core;
- the "single pipeline" mode - allows you to increase core's life in critical environments. This mode is integrated and supported by the hardware but requires that the code is written ad properly.

A portion of the software is used to check the correct functioning of the hardware: if a pipeline is damaged, it can be disabled. The core is then used with a single unprotected pipeline. The robustness of the processing must be granted to the software, which will be executed in a redundant manner, sacrificing the processing speed.

The CRU is the heart of the DoublePipe architecture protection system. The system is based on the comparison of the pipeline outputs. In case of output's discrepancy, the CRU activates a flag that indicates the presence of an error. In the next execution phase of the thread with an active flag, the CRU takes control of the outputs, simulating an illegal instruction with a specific cause code. At this point, a software routine takes care of recovering the values of the register files previously saved in memory. At the end of the illegal instruction routine, the PC unit loads the program counter with the value saved during checkpoint creation. The return to a checkpoint does not deactivate the protected mode or eliminate the checkpoint. The unit manages part of the dedicated instructions of this architecture and the internal control register. The control register contains information on the configuration of the CRU, on the execution status of the core and on any hanging errors. This register can be read (but not written) by the user.

The DuoblePipe CSR, unlike the original version, includes a specific register which is used to back up the PC. The register is not addressable, and its writing is managed by the CSR during the execution of the pseudo instructions developed for this architecture. Other registers are instead extended in use and functionality compared to the RISC-V standard: the writing with particular values of some registers will be interpreted as an instruction. The CSR, together with the CRU, takes care of serving the instructions for starting a checkpoint and restoring it in the event of an error.

The DoublePipe program counter unit is substantially identical to the original version in terms of functionality. Since the activation system of a checkpoint is based on the start of a particular software interrupt, it is necessary to add a condition in the PC unit that allows the service of this type of interrupt despite the interrupts being disabled.

7.4 F03c: Shadow Thread Double Pipe

The F03c architecture is based on the possibility to correct errors, by using a double pipeline and a redundant execution of the thread instructions. To obtain corrections of the errors, 3 copies of the same result are necessary:

- two contemporary copies obtained through pipeline redundancy;
- a third copy obtained through a temporally out of phase processing.

A Shadow control unit (SCU) handle the pipelines and the architecture synchronization. During the shadow processing, the SCU handles two different instructions. These are executed in the pipelines. To solve latency problem between instructions, the SCU can put the pipeline on hold in order to complete the execute phase in a synchronized way. The register file management is left to SRU unit, that constantly communicates with the SCU. The SCU provides information about processing in the pipeline, indicating to the SRU whether the instructions require access to the registers. In case of error, the SRU performs a memory access and retrieves the value of the register. If the error is detected in conjunction with an instruction that requires reading the same file, the SRU sends a signal to the SCU which blocks the pipelines. Once received the data from the memory, a triple comparison is made, and the correct data is sent to the pipelines. When this phase is completed, the SCU unlocks the pipelines. The writing in memory is started in conjunction with the WB phase. The copy of the regfile has priority over a possible access instruction in memory, which is put on hold, locking the pipelines.

The CSR ST does not contain dedicated registers but is equipped with additional input signals. The operation of the unit depends on:

- main processing: the CSR executes the commands received from the pipeline processing, previously controlled by the SCU. It also reacts to any hardware routines to serve exceptions and interrupts or following a return instruction.
- Shadow processing: the CSR simulates the execution of the access instructions by supplying the values contained in the registers. The values contained in the registers are not modified unless explicitly commanded by the SCU. No interrupt or exception affects the CSR at this step. An eventual interrupt event is served at the next main processing.

Writing to the internal registers can be disabled at any time by the SCU, which keeps a constant check on the CSR. This architectural difference allows the value sent to the registers to be blocked at any time. Loading incorrect values (in the TMR registers) is always prevented. The triple redundancy technique completely loses its effectiveness in the event of an error in the logic that sends the data.

The architecture of the PC ST unit differs from the original version of the core as it must guarantee the functioning of the shadow structure that requires up to two PCs simultaneously. The portion that manages the PC during the fetching phase of the shadow processing is located inside the SCU. The PC ST unit, in addition to supplying the correct PC value to the SCU, must manage the correct updating of the internal PC registers. This is done by receiving information on the location of the shadow thread within the Pipeline. Thanks to this information the unit can execute or block the updating of the PC registers. In the event of interrupts, exceptions or jumps during the execution phase of the Shadow processing, the unit locks the PC update system waiting for the main processing phase. If the conditions that triggered an interrupt, an exception or any request to change the program flow remain, the PC will update and start the procedure. This allows to avoid serving the same interrupt twice. In this unit there is an input signal that allows the SCU to stop updating the PC at any time in case of error. In this case indeed, the TMR protection of the inside registers is not able to correct the loading of an incorrect value.

Chapter 8

Debug Support

Klessydra core supports common baseline debug features: halting the program flow, reading data register file, reading the PC value and enabling a single step execution. Software breakpoints are implemented by the RISC-V instruction EBREAK.

The debug operations are intended at core level and not per-thread. When entering debug mode, the whole core (i.e. with all its threads) enters debug mode. The internal debug unit accesses information related to the thread whose instruction is in the execution stage of the core pipeline in the current clock cycle.

The debug hardware interface is the same as the memory interface, but on separate buses. Every access to debug facilities is done by an access to debug registers.

To halt the core, external debug unit has to set DBG_CTRL[0] bit. If DBG_CTRL[0] is set, the core is in single step mode, so clearing the DGB_HIT[0] bit enable execution of a single instruction.

Debug registers are always accessible. Program counter and register file are accessible only when the core is halted. Which register of register file external debug unit requires is specified in [6:2] bit of the address.

Table.8.1 Debug Registers

Address	Name	Description
0x00	DBG_CTRL	Debug Control
0x04	DBG_HIT	Debug Hit
0x2000	DBG_PPC	Next PC
0x2004	DBG_NPC	Previous PC
0x400-0x47C	GPR(x0-x31)	General Purpose Registers

Table.8.2 Debug Control register bit map

Bit #	R/W	Description
16	R/W	HALT bit: When set to '1', the core enters debug mode, when reset to '0', the core exits debug mode.
0	R/W	SSTE bit: Single-step enable bit.

Table.8.3 Debug Hit register bit map

Bit #	R/W	Description
0	R/W	SSTH: Single-step hit, sticky bit that must be cleared by external debugger in order to execute next instruction.

Table.8.4 Debug Next Program Counter register bit map

Bit #	R/W	Description
31:0	R/W	NPC: Next PC to be executed

Table.8.5 Debug Previous Program Counter register bit map

Bit #	R/W	Description
31:0	R/W	PPC: Previous PC, already executed

Chapter 9

Instruction Set

9.1 Integer Register-Immediate operations

Table.9.1 Register-Immediate operations

Name	Binary format type	Assembly syntax
ADDI – add immediate	I	ADDI rd, rs1, imm
SLTI - set if less immediate	I	SLTI rd, rs1, imm
SLTIU - set if less imm. uns.	I	SLTIU rd, rs1, imm
ANDI - and immediate	I	ANDI rd, rs1, imm
ORI - or immediate	I	ORI rd, rs1, imm
XORI – excl. or immediate	I	XORI rd, rs1, imm
SLLI – shift left logical imm.	I	SLLI rd, rs1, shamt
SRLI– shift right logical imm.	I	SRLI rd, rs1, shamt
SRAI – shift right arithm. imm.	I	SRAI rd, rs1, shamt
LUI - load upper immediate	U	LUI rd, imm
AUIPC - add upper imm. to pc	I	AUIPC rd, imm

- ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low 32 bits of the result. `ADDI rd, rs1, 0` can be used to implement a register move operation.
- SLTI places the value 1 in register *rd* if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers.
- ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Notably, `XORI rd, rs1, -1` performs a bitwise logical inversion of register *rs1*.
- SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits). The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field.
- LUI is used to build 32-bit constants. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.
- AUIPC is used to build PC-relative addresses. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the PC, then places the result in register *rd*.

9.2 Integer Register-Register Operations

Table.9.2 Register-Register Operations

Name	Binary format type	Assembly syntax
------	--------------------	-----------------

ADD - add	R	ADD rd, rs1, rs2
SLT - set if less	R	SLT rd, rs1, rs2
SLTU – set if less unsigned	R	SLTU rd, rs1, rs2
AND - and	R	AND rd, rs1, rs2
OR - or	R	OR rd, rs1, rs2
XOR - exclusive or	R	XOR rd, rs1, rs2
SLL – shift left logical	R	SLL rd, rs1, rs2
SRL – shift right logical	R	SRL rd, rs1, rs2
SUB – subtract	R	SUB rd, rs1, rs2
SRA - shift right arithmetic	R	SRA rd, rs1, rs2

- ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low 32 bits of results are written to the destination.
- SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if $rs1 < rs2$, 0 otherwise. Note, SLTU *rd, x0, rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudo-op SNEZ *rd, rs*).
- AND, OR, and XOR perform bitwise logical operations.
- SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

9.3 Unconditional Jumps

Table.9.3 Unconditional Jumps

Name	Binary format type	Assembly syntax
JAL - jump and link	UJ	JAL rd, imm
JALR – jump to reg and link	UJ	JALR rd, rs1, imm

- The jump and link (JAL) instruction uses the J-immediate to encode a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following the jump (PC+4) into register *rd*. Plain unconditional jumps are encoded as a JAL with *rd* = *x0*.
- The indirect jump instruction JALR (jump and link register) obtains the target address by adding the 12-bit signed I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (PC+4) is written to register *rd*. Register *x0* can be used as the destination if the result is not required.
- The JAL and JALR instructions will generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary.

9.4 Conditional Branches

Table.9.4 Branches

Name	Binary format type	Assembly syntax
BEQ – branch if equal	SB	BEQ rs1, rs2,imm
BNE - branch if not eq.	SB	BNE rs1, rs2,imm
BLT – branch if less	SB	BLT rs1, rs2,imm
BGE– branch if greater	SB	BGE rs1, rs2,imm
BLTU – branch if less	SB	BLTU rs1, rs2,imm
BGEU – branch if greater	SB	BGEU rs1, rs2,imm

- BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively.
- BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively.
- BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively.
- All branch instructions use the 12-bit B-immediate to encode signed offsets in multiples of 2, and add the offset to the current PC to give the target address. The conditional branch range is ± 4 KiB.

9.5 Memory access Instructions

Table.9.5 Load-Store Instructions

Name	Binary format type	Assembly syntax
LB - load byte	I	LB rd, rs1, imm
LH - load half word	I	LH rd, rs1, imm
LW - load word	I	LW rd, rs1, imm
LBU - load byte unsigned	I	LBU rd, rs1, imm
LHU - load half word unsig.	I	LHU rd, rs1, imm
SB - store byte		SB rs1,rs2,imm
SH - store half word		SH rs1,rs2,imm
SW - store word		SW rs1,rs2,imm

- Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.
- The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory

9.6 CSR Instructions (Read-Set-Clear)

Table.9.6 CSR Instructions

Name	Binary format type	Assembly syntax
CSRRW - csr read/write		CSRRW rd, csr, rs1
CSRRS - csr read & set		CSRRS rd, csr, rs1
CSRRC - csr read & clear		CSRRC rd, csr, rs1
CSRRWI - csr rd/wr. Imm.		CSRRWI rd, csr, imm
CSRRSI - csr rd & set imm		CSRRSI rd, csr, imm
CSRRCI - csr rd & clr imm		CSRRCI rd, csr, imm

- The CSRRW instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to 32 bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If $rd=x0$, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.
- The CSRRS instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).
- The CSRRC instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.
- For both CSRRS and CSRRC, if $rs1 \neq x0$, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if rs1 specifies a register holding a zero value other than x0, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects.
- The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an 32-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if $rd=x0$, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

9.7 CSR Privileged Instructions

Table.9.7 Privileged Instructions

Name	Binary format type	Assembly syntax
ECALL – environment call		ECALL
EBREAK – break to envir.		EBREAK
WFI – wait for IRQ		WFI
MRET – machine return		MRET

- The ECALL instruction is used to make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.
- The EBREAK instruction is presently implemented in the S0 core only (future update in T0 cores and T1 cores).
- The WFI is a wait for interrupt instruction, that latches the thread in an idle state until an interrupt arrives.
- The MRET updates the program counter with the address of the instruction being executed before entering the trap handling routine. Unless the instruction was a WFI, we return to the address after it.

9.8 Atomic Instructions

Table.9.8 Atomic Instructions

Name	Binary format type	Assembly syntax
AMOSWAP.W.AQ	R	AMOSWAP.W.AQ rd,rs1,rs2
AMOSWAP.W.RL	R	AMOSWAP.W.RL rd,rs1,rs2

- The atomic memory operations AMOSWAP.W atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a swap between the loaded value and the original value in *rs2*, then store the swapped value to the address in *rs1*. The implementation follows “release consistency”. The AMOSWAP.W.AQ instruction implements a read-modify-write operation suited to lock acquiring, while the AMOSWAP.W.RL instruction implements a read-modify-write operation suited to lock releasing.
The S0 core does not support Atomic Instructions.

9.9 Klessydra Custom Extensions (T1 version only)

Table.9.9 Klessydra custom extensions

Name	Binary format type	Assembly syntax
KMEMLD	R	kmemld rd, rs1, rs2
KMEMSTR	R	kmemstr rd, rs1, rs2
KADDV	R	kaddv rd, rs1, rs2
KSUBV	R	ksubv rd, rs1, rs2
KVMUL	R	kvmul rd, rs1, rs2
KVRED	R	kvred rd, rs1, rs2
KDOTP	R	kdotp rd, rs1, rs2
KSVADDSC	R	ksvaddsc rd, rs1, rs2
KSVADDRF	R	ksvaddrf rd, rs1, rs2
KSVMULSC	R	ksvmulsc rd, rs1, rs2
KSVMULRF	R	ksvmulrf rd, rs1, rs2

KDOTP	R	kdotp rd, rs1, rs2
KDOTPPS	R	kdotpps rd, rs1, rs2
KSRLV	R	ksrlv rd, rs1, rs2
KSRAV	R	ksrav rd, rs1, rs2
KRELU	R	krelu rd, rs1
KVSLT	R	kvslt rd, rs1, rs2
KSVSLT	R	kvsslt rd, rs1, rs2
KBCAST	R	kbcast rd, rs1
KVCP	R	kvcp rd, rs1

- KMEMLD: loads the number of bytes specified by 'rs2' in the scratchpad memory at address 'rd', from the address 'rs1' in the main memory.
- KMEMSTR: loads the number of bytes specified by 'rs2' in the main memory at address 'rs1', from the address 'rd' in the scratchpad memory.
- KADDV: adds the operands in the scratchpad at addresses in 'rs1' and in 'rs2' and stores the result as a vector at the address 'rd' in the scratchpad memory.
- KSUBV: subtracts the operands in the scratchpad at addresses in 'rs1' and in 'rs2' and stores the result as a vector at the address 'rd' in the scratchpad memory.
- KVMUL: multiplies the vector elements of rs1 and rs2 and stores the result in rd.
- KVRED: performs vector reduction between the elements at addresses 'rs1' and 'rs2', and stores the scalar in 'rd'.
- KDOTP: multiplies the operands at addresses in 'rs1' and in 'rs2', the multiply intermediate results are accumulated, and the final results are stored as a scalar in the address in 'rd'.
- KDOTPPS: performs post scaling dot product on the elements at addresses in 'rs1' and 'rs2' and puts the result in 'rd'. The multiplication result is shifted by the value set the CSR register 'MPSCLFAC'.
- KSVADDSC/RF: adds the scalar operand in the register file or scratchpad address in 'rs1' with a scalar value that is in 'rs2'. The result is stored as a vector at address in 'rd'. (A faster alternative to using KBCAST).
- KSVMULRF/SC multiplies the scalar operand in the register file / scratchpad in 'rs1' with a scalar value that is in 'rs2'. The result is stored as a vector in the address in 'rd'. (A faster alternative to using KBCAST).
- KSRLV/KSRAV: does right logical/arithmetic shifts on the vector at the address in 'rs1' by the shift amount in 'rs2' and stores the vector results at the address in 'rd'.
- KRELU: does linear rectification on the negative values of the vector at the address in 'rs1' and puts the rectified vector at the address in 'rd'.
- KVSLT: Is a vector set less than used to create masks, that is done when two vectors in rs1 and rs2 are compared element by element such that when " $\text{vect1}[i] < \text{vect2}[i]$ " we set $\text{vect_result}[i] = '1'$, else '0'.
- KSVSLT: Is a vector set less than used to create masks, that is done when a vector in rs1 is compared to a regfile-scalar in rs2 such that when " $\text{vect1}[i] < \text{scal2}[i]$ " we set $\text{vect_result}[i] = '1'$, else '0'.
- KBCAST: does a vector broadcast of the scalar value contained in scalar register 'rs1' to the vector at the address in 'rd'.
- KVCP: copies the vectors starting at the address in 'rs1' to the address in 'rd'. Both addresses are in scratchpad memory space.

All logical-arithmetic vector instructions should all be used in conjunction with the CSR register 'MVSZIE' in order to specify the size of the vector to be processed by the operation.

Glossary

CSR = Control and Status Registers

Harc = (hardware context) a positive integer number identifying a thread in the processing core.

SPM = Scratchpad memory

IRQ = interrupt request

Klessydra = the name of the family of processing cores reported in this manual.

PULP = an open-source multi-core processor architecture

PULPino = an open-source System-on-Chip single-core microcontroller architecture

S0 = a core belonging to the Klessydra family featuring single-thread execution at minimum hardware cost

T0 = a set of cores belonging to the Klessydra family, supporting interleaved multiple thread execution

T1= enhanced version of the T0 cores designed to widen the target applications of Klessydra for vector applications