Klessydra Lightweight Runtime System

Technical Manual

(Preliminary edition – July 2020)

SAPIENZA
UNIVERSITÀ DI ROMA

Klessydra Lightweight Runtime System

Technical Manual

Release Information

Jul. 2020 preliminary edition

Project Status

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

Contents

Preface

About This Manual

This manual illustrates the Klessydra processor core software library constituting the processors' runtime system.

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

Related documents

Klessydra Processing Core Family Technical Manual, Online: http://vlsi.diet.uniroma1.it/downloads/KlessydraTechnicalManual.pdf

A.Traber, M. Gautschi, *Pulpino: Datasheet*, 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:

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

Chapter 1
Introduction

The Klessydra Lightweight Runtime System for operation within the Pulpino platform, is composed of the following files:

- klessydra.h, containing macros and constant values necessary to the operation of the runtime system;
- pulpino.h, that includes klessydra.h and contains the original basic definitions necessary to the operation of the Pulpino platform;
- crt0.klessydra.S, containing the assembly-coded startup routine and interrupt management routine, compatible with the Pulpino platform. These routines are transparent to the user;
- crt0.klessydra_E.S, performing the same function as the orgina startup code, but written for when the Klessydra is configured to run as the Embedded RV32E extension.
- klessydra_defs.h, containing an elementary C function library for thread management.
- dsp_functions.h, containing a set of inline functions used to run computations through the vector accelerator, as well as having other various functions.

When the Klessydra cores are used as stand-alone or in a platform different from Pulpino, the file crt0.klessydra.S/crt0.klessydra_E.S should be changed with the version reported in the directory kless/stand_alone_startup of the Klessydra distribution.

This manual addresses the runtime system for operation within the Pulpino platform.

Chapter 2

crt0.klessydra.S

2.1 General description

The crt0.klessydra.S/ crt0.klessydra_E.S file is the Pulpino/Klessydra startup file to be used for operation of klessydra cores within the Pulpino platform.

In these files, there are two important sections, "Stack initilization" and "trap handler" (i.e "MTVEC routine"). The former the first routine that Klessydra core executes at boot.

2.2 Stack initialization routine

This routine allocates a stack pointer (*sp*) for each thread and a global pointer (*gp*) for the remaining free space.

In order to compile any software application it is necessary that the header file pulpino/sw/libs/sys_lib/inc/klessydra.h contains the definition the desired stack size by means of the macro

```
#define thread_stack_size 4096  // user defined value
```

The maximum number of threads that can be used are 16, so that the routine is not able to allocate more than 16 stack pointers.

When choosing the "thread_stack_size" one must ensure not to overflow the data memory size.

2.3 Trap handler routine

Klessydra solves the trap cause via a software routine pointed to by the MTVEC register of the trapped thread (MTVEC routine). In other words, the MTVEC routine is the software routine the thread jumps to each time it must handle a trap. It operates as follows:

1) It compares the MCAUSE register content with the default trap codes, untill it finds the trap cause.
2) If the trap cause is an external interrupt, it jumps to the address contained in the MIRQ register.
3) If the trap cause isn't an external interrupt, it continues to compare MCAUSE register searching for other cause codes.
4) If it doesn't find any coded trap cause, it performs an MRET instruction (this should never happen in correct operation).

When the routine identifies the trap cause, it jumps to a PULPino trap handler which serves the trap.

Each PULPino trap handler always contains two subroutines, store_regs and end_except, that Pulpino performs before and after serving the interrupt. The store_regs subroutine saves all caller-saved registers (except the return address) so that we don't overwrite the stack while serving the interrupt. The end_except subroutine loads back the registers from the stack in order to restore the state of the trapped thread.

The Pulpino software suite does not implement any pre-defined routine for the following Klessydra specific trap causes:

- *SW_INTERRUPT;*
- *LOAD_ERROR_EXCEPT;*
- *STORE_ERROR_EXCEPT;*
- *LOAD_MISALIGNED_EXCEPT;*
- *STORE_MISALIGNED_EXCEPT*.

For the above causes a dedicated routine should be provided by the user according to the target application.

Chapter 3
Functions.h

3.1 Requirements

Before using the following functions you must define the maximum number of threads that the core core can run, i.e. the thread pool size. This is done by setting the following macro at line 9 of the file klessydra_threads.h:

```
#define THREAD_POOL_SIZE <number>;
```

3.2 Klessydra_get_mhartid()

This function returns the MHARTID register value.
The MHARTID value includes the coreID and clusterID according to the PULP platform definitions.

3.3 Klessydra_get_coreID()

This function returns the coreID. This return value of this function is often utilized in order to let a thread with a specific coreID execute a scertain subroutine.

3.4 Klessydra_En_Int()

This is a void function that enables interrupt handling by writing to the MSTATUS CSR register MIE bit. With the MIE bit of the hart set to ‘0’ the core will not be able to handle interrupts.

3.5 klessydra_lock_acquire(int *plock)

This function allows one to acquire a lock prior to enter a specific code section (critical section).

One can use this function to be sure that only one thread is working with the instructions inside the critical section.

This function uses a global variable that must be declared and initialized to zero:and substitutes that ‘0’ with a lock.

3.6 klessydra_lock_release(int *plock)

This function allows you to release a lock on a specific code section. Only unlocked threads can execute this function to release the locked ones.

3.7 send_sw_irq(int target_hart)

This function allows you to send a software interrupt to a specific thread.

The target_hart argument is the coreID of the thread to which the software interrupt will be sent.

The function returns 0 if the thread to which you try to sent the software interrupt doesn't exist, it returns 1 if the thread exists.

This functions is often used to release the harts from a waot for interrupt state *“WFI”.*

3.8 Synchronization barrier functions

These functions are used to set a synchronization point on which all the threads pause on a *WFI*, until all threads previously registered to that barrier arrive. Then, they all restart together and resume executing their code.

To use these features you must initialize the barrier using the function at least once.

```
sync_barrier_reset();
```

To register a thread on the barrier can be done with using this function:

```
sync_barrier_thread_registration();
```

You must insert this function in a code section executed by the thread you want to sign in.
To insert the breakpoint, in which the threads will be resynchronized you have to use this function:

`sync_barrier()`

Be sure that all the threads which have registered themselves have entered this function.

All the threads which will perform the `sync_barrier()` function will stay in an idle state untill the last one arrives and sends a software interrupt to wake up the others..

Chapter 4
DSP_Functions.h

4.1 Requirements

These functions can only be used with the Klessydra-T13 processor, which contains a highly parameterizable vector coprocessor.

4.2 DSP-Conf-Functions

4.2.1 CSR_MVSIZE()

Takes as an input the size of the vector in bytes. The maximum size cannot exceed the SPM size.

4.2.2 CSR_MVTYPE()

Takes as an input the data type. An input of ‘0’ sets the elements to 8-bit ints, ‘1’ for 16-bit ints, and ‘2’ for 32-bit ints.

4.2.3 CSR_MPSCLFAC()

Take as input the post-scaling factor used to right shift results by a fixed amount after a multiplication operation is performed.

4.3 DSP-Mem-Functions

4.3.1 kmemld()

Takes the following form, kmemld(void* rd, void* rs1, int rs2). This function loads ‘rs2’ bytes from the data-mem pointer in ‘rs1’ and loads the bytes in ‘rd’ SPM address.

4.3.2 kbcastld()

Takes the following form, kbcastld(void* rd, void* rs1, int rs2). This function loads ‘rs2’ bytes from the pointer in ‘rs1’ and loads the bytes in ‘rd’ SPM address of each hart. The way this function differs from the kmemld, is that kmemld loads the bytes into the SPM of the hart it belongs to, while a hart executing kbcastld loads the data into all the SPMs of the harts.

Note: If vector coprocessor is not replicated per hart, then kmemld, and kbcastld perform exactly the same function.

4.3.3 kmemstr()

Takes the following form, kmemstr(void* rd, void* rs1, int rs2). This function loads ‘rs2’ bytes from the SPM pointer in ‘rs1’ and stores the bytes in ‘rd’ data-mem address.

4.4 DSP-Arith-Functions

Note 1: Functions with _v2 suffix provide and additional argument to set the vector size.
Note 2: Functions with _v3 suffix take a post scaling factor to right shift computations by a fixed amount.
Note 3: Functions with _v4 suffix take both arguments from _v2 and _v3 functions.
Note 4: All the functions work with with the different vtypes set in the MTYPE CSR reg.

4.4.1 kaddv() / kaddv_v2

Takes the following form, kaddv(void* rd, void* rs1, void* rs2). This function adds the elements in the SPM addresses ‘rs1’ and ‘rs2’ and stores the result in SPM address ‘rd’.

4.4.2 ksvaddrf() / ksvaddrf_v2

Takes the following form, ksvaddrf(void* rd, void* rs1, void* rs2). This function adds the elements in the SPM address in ‘rs1’ with regfile scalar in ‘rs2’ and stores the result in SPM address ‘rd’. Note: ‘rs2’ is not a pointer.

4.4.3 ksvaddsc() / ksvaddsc_v2

Takes the following form, ksvaddsc(void* rd, void* rs1, void* rs2). This function adds the elements in the SPM address in ‘rs1’ with SPM scalar in ‘rs2’ and stores the result in SPM address ‘rd’. Note:’rs2’ is a pointer.

4.4.4 ksubv() / ksubv_v2

Takes the following form, ksubv(void* rd, void* rs1, void* rs2). This function subtracts the elements in the SPM addresses ‘rs2’ from the elements in ‘rs1’ and stores the result in SPM address ‘rd’.

4.4.5 kvmul() / kvmul_v2

Takes the following form, kvmul(void* rd, void* rs1, void* rs2). This function multiplies the elements in the SPM addresses ‘rs1’ and ‘rs2’ and stores the result in SPM address ‘rd’.

4.4.6 ksvmulrf() / ksvmulrf_v2

Takes the following form, ksvmulrf(void* rd, void* rs1, void* rs2). This function multiplies the elements in the SPM address in ‘rs1’ with regfile scalar in ‘rs2’ and stores the result in SPM address ‘rd’. Note: ‘rs2’ is not a pointer.

4.4.7 ksvmulsc() / ksvmulsc_v2

Takes the following form, ksmulsc(void* rd, void* rs1, void* rs2). This function multiplies the elements in the SPM address in ‘rs1’ with SPM scalar in ‘rs2’ and stores the result in SPM address ‘rd’. Note:’rs2’ is a pointer.

4.4.8 ksvmuladd() / ksvmuladd_v2

Takes the following form, ksvmuladd(void* rd, void* rs1, void* rs2, void* rs3). This function performs a ksvmulsc between the vector in ‘rs2’ and the scalar in ‘rs3’ and puts the result in ‘rd’. Then a kaddv adds the result in ‘rd’ with the vector in ‘rs1’ overwriting the previous result in ‘rd’.

4.4.9 kdotp() / kdotp_v2

Takes the following form, kdotp(void* rd, void* rs1, void* rs2). This function does dot-product between the vectors in the SPM addresses ‘rs1’ and ‘rs2’ and stores the scalar result in SPM address ‘rd’.

4.4.10 kdotpps() / kdotpps_v2 / kdotpps_v3 / kdotpps_v4

Takes the following form, kdotpps(void* rd, void* rs1, void* rs2). This function performs the same behavior as kdotp between the vectors in the SPM addresses ‘rs1’ and ‘rs2’ and stores the scalar result in SPM address ‘rd’. However after the multiplication and right before the accumulation, the mul results will be pre-scaled by a shift amount defined in MVTYPE.

4.4.11 kdotpps_emul_v3() / kdotpps_emul_v4

Takes the following form, kdotpps_emul_v3(void* rd, void* rs1, void* rs2, void* p_scal). This function emulates the *kdotpps* instruction by using the *kvmul*, *ksrav*, and *kvred* instructions.

4.4.12 kvred() / kved_v2

Takes the following form, kvred(void* rd, void* rs1). This function performs vector reduction by accumulating the elements in the vector in the SPM address ‘rs1’ stores the scalar result in SPM address ‘rd’.

4.4.13 ksrav() / ksrav_v2

Takes the following form, ksrav(void* rd, void* rs1, void* rs2). This function performs a vector arithmetic right shift of the vector in the SPM address ‘rs1’ by the regfile scalar shift amount in ‘rs2’ and stores the scalar result in SPM address ‘rd’. Note: ‘rs2’ is not a pointer.

4.4.14 ksrlv() / ksrlv_v2

Takes the following form, ksrlv(void* rd, void* rs1, void* rs2). This function performs a vector logic right shift of the vector in the SPM address ‘rs1’ by the regfile scalar shift amount in ‘rs2’ and stores the scalar result in SPM address ‘rd’. Note: ‘rs2’ is not a pointer.

4.4.15 krelu() / krelu_v2

Takes the following form, krelu(void* rd, void* rs1). This function performs relu operation which rectifies all the negative values in the vector in the SPM address ‘rs1’ and stores the result in SPM address ‘rd’.

4.4.16 kvslt() / kvslt_v2

Takes the following form, kvslt(void* rd, void* rs1, void* rs2). This function creates a mask, by comparing the vectors in SPM address ‘rs1’ and ‘rs2’ and writes ‘1’ if vec_1(i) < vec_2(i) else ‘0’. The vector mask is stored in SPM address in ‘rd’.

4.4.17 ksvslt() / ksvslt_v2

Takes the following form, ksvslt(void* rd, void* rs1, void* rs2). This function creates a mask, by comparing the vector in SPM address ‘rs1’ with the regifle scalar in ‘rs2’ and writes ‘1’ if vec_1(i) < scal else ‘0’. The vector mask is stored in SPM address in ‘rd’.

4.4.18 kbcast() / kbcast_v2

Takes the following form, kbcast(void* rd, void* rs1). This function broadcasts the regifle scalar in ‘rs1’ to the vector of size MVSIZE in SPM address in ‘rd’.

4.4.19 kvcp() / kvcp_v2

Takes the following form, kvcp(void* rd, void* rs1). This function copies the vector in the SPM address in ‘rs1’ to the SPM address in ‘rd’.

Note: Other functions are available for debugging purposes, and their description can be found in the header file itself.