Imperas ARM Processor Model Application Note

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	0.1
Filename:	Imperas_ARM_Processor_Model_Application_Note.doc
Project:	ARM Processor Model Application Note
Last Saved:	February 22, 2021
Keywords:	OVP ARM Processor MultiProcessorOVP ARM Processor MultiProcessor

Table of Contents

1 Preface

This document describes the ARM Multi-processor architecture and how this is used and modeled in the OVP modeling environment.

1.1 Related Documents

The reader should be familiar with Open Virtual Platforms (OVP) and the ARM architecture.

1.2 Trademarks

ARM®, Cortex™, MPCore™ and any other trademark found on the ARM trademarks list that are referred to or displayed in this document are trademarks or registered trademarks of ARM Ltd or its subsidiaries.

Imperas acknowledge trademarks or registered trademarks of other organizations for their respective products and services.

2 ARM Multiprocessor Configuration

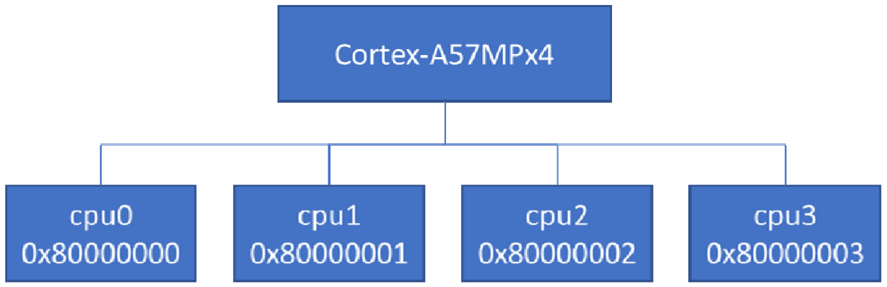
Configuration of what a processing element (PE) is in an ARM core or cluster is defined by the MPIDR system register. The format of this is as follows (for AArch64):

63 40	39 32	31	30	29 25	24	23 16	15 8	7 0
RES0	Aff3	RES1	U	RES0	MT	Aff2	Aff1	Aff0

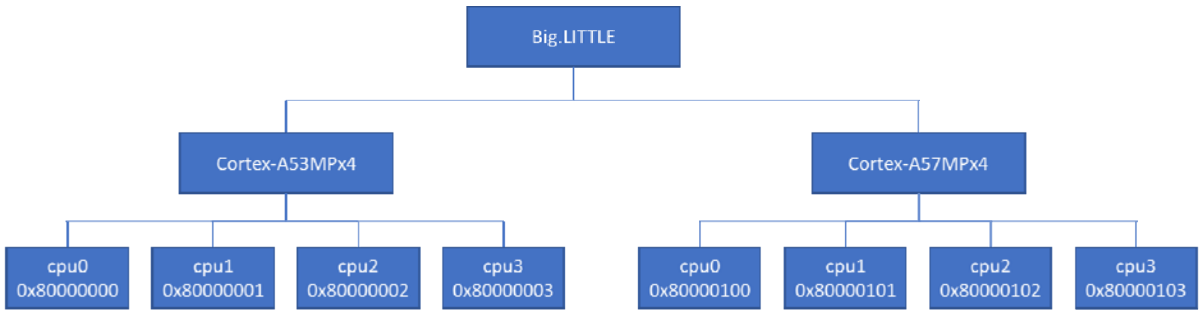
This register has a different value for each processing element in the system. Byte-sized fields Aff0, Aff1, Aff2 and Aff3 give the core *affinity level*, in lowest to highest order; Aff0 distinguishes the most tightly coupled PEs. Field U indicates the core is part of a uniprocessor system if non-zero. Field MT specifies whether the lowest level of affinity consists of logical PEs that are implemented using a multithreading type approach: this is 0 if the cores are independent and 1 if they are threaded.

For a uniprocessor PE, MPIDR will be zero except for bit 31 and the U bit which will be 1.

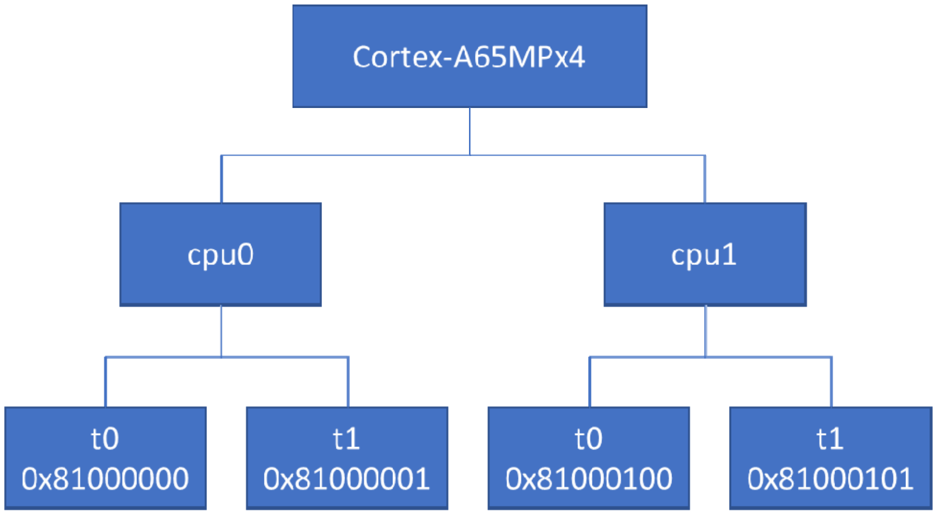
For PEs in something like a Cortex-A57MPx4, MPIDR values will be zero except for the Aff0 field, which will be 0 for PE0, 1 for PE1 etc. When we instantiate a Cortex-A57MPx4 in the simulator, we therefore create a single root level object containing 4 PEs with these values of MPIDR:



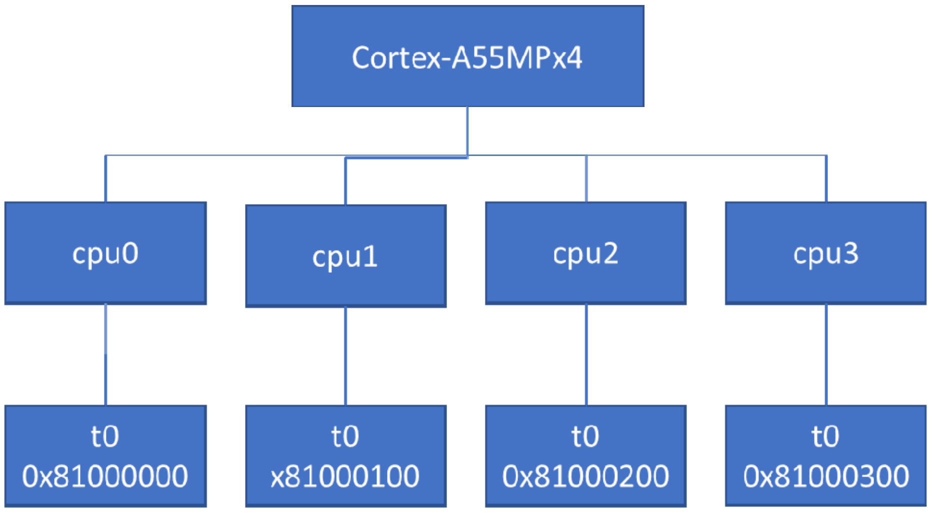
The rule in the simulator is that an extra level of hierarchy is added out to that highest Aff* that varies. In this case, only Aff0 varies, so one extra hierarchy level is added. For PEs in a big/little type cluster, affinity level Aff1 is used to distinguish the big/little halves. When we instantiate a cluster like this, the root therefore contains two cores, each with multiple PEs:



The traditional ARM cores modeled are not multithreaded (MPIDR.MT=0). More recent cores, starting with the Cortex-A55 are logically *multithreaded* (MPIDR.MT=1). In such cores, the lowest affinity level (Aff0) indicates cores that are tightly coupled threads that share resources. For example, in a Cortex-A65, each core is dual-threaded, so Aff0 can be 0 or 1. We don't yet have a Cortex-A65 model, but if we did then a Cortex-A65MPx4 would look like this:



The Cortex-A55 is a special *degenerate* case: it is logically *multithreaded* (MPIDR.MT=1) but physically *single threaded*, because Aff0 is forced to 0 on every PE. When we instantiate a Cortex-A55MPx4, we therefore get this hierarchy:



(Aff1 is the highest level that varies, so there are two extra hierarchy levels.) I don't know why ARM decided to make the A55 look like this instead of having it follow the standard pattern of (for example) an A53, but I guess that it was a stepping-stone towards *physically* multithreaded cores like the A65. From our perspective, because the instantiation hierarchy is driven from the

MPIDR register, the effect is that there is an extra level of processor object in the hierarchy. This has caught out several customers in the past so it is something to be aware of.

3 Simulation Considerations

The Imperas simulation algorithm tries to handle multithreaded cores specially to give an impression of the resource constraints they imply. For threads, these things are typically shared in the hardware:

- caches
- execution units (ALU, floating point unit)
- store buffers
- branch table entries
- some signals (e.g. perhaps reset signal)

What this means is that if two threads are using these units heavily at the same time, they will both run slower because of resource contention. To model this, the Imperas default scheduler handles threaded cores using a *micro-scheduler:* when a platform containing a threaded CPU is simulated, the quantum's-worth of instructions for that threaded core are further broken down into sub-quantums (the number of which is specified when the processor instance is created) and dispatched to the runnable threads on the core in a round-robin fashion. If a threaded core contains (say) 2 threads and a quantum has 1000 instructions based on the processor MIPS rate, then each thread would therefore execute 500 instructions. However, if one of the threads is halted, then the other would execute the full 1000 instruction allotment. This gives some impression of the effect of resource conflict during simulation.

When using `opProcessorSimulate`, it is possible to call *either* on the threaded CPU *or* on the individual threads, depending on the effect desired. If the call is made on the threaded CPU, then the micro-scheduling algorithm briefly described above is applied on the contained threads. If called on the individual thread, then that thread alone runs, in the "normal" way. One important point is that, within a threaded CPU, *all threads have a common notion of local time*. In other words, if the first thread simulates for a number of instructions then the second thread will definitely see state as if time has moved on by that number of instructions.

One problem is with external simulation environments like SystemC. SystemC expects (requires?) each runnable processor context to be in a separate native thread (so that it can model delays by halting the thread in a memory callback, for example). This means that it can't use the micro-scheduler simulation level, because a blocking memory access on one thread would also halt the other. To handle simulation in such environments, the ARM model has a parameter, `distinctMTCores`, which disables micro-scheduling and forces all leaf threads to be treated as independent execution entities. For processors with multiple threads at leaf-level being used in SystemC environments, always set this parameter. For the Cortex-A55 it should not be necessary to set this parameter because there is only one thread at each leaf level.

##