VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ASYMETRICKÝ MULTIPROCESING NA ARM CORTEX – A9

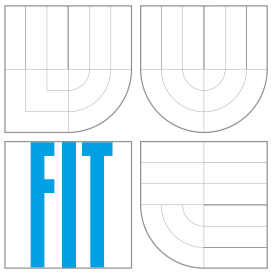
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL RIŠA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

ASYMETRICKÝ MULTIPROCESING NA ARM CORTEX – A9

ASYMMETRIC MULTIPROCESSING ON THE ARM CORTEX – A9

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL RIŠA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KORČEK PAVOL

BRNO 2015

Abstrakt

Asymetrický multiprocessing (*AMP*) je způsob rozdělování zátěže počítačového systému na heterogenní hardwarové a softwarové prostředí. Tato práce popisuje principy *AMP* se zaměřením na ARM Cortex–A9 procesor a Altera Cyclone V hardwarovou platformu. Postup tvorby *AMP* systému založeného na OpenAMP frameworku ukazujícího komunikaci mezi procesorovými jádry, dokumentace a prognóza budoucího vývoje jsou výstupy této práce.

Abstract

Asymmetric multiprocessing (*AMP*) is a way of distributing computer system load to heterogeneous hardware and software environment. This thesis describes the principles of the *AMP* focusing on the ARM Cortex–A9 processor and Altera Cyclone V hardware platform. Development of a OpenAMP framework based *AMP* system showing communication among the processor cores, documentation and future work suggestion are the products of this thesis.

Klíčová slova

AMP, SMP, SoC, FPGA, ARM Cortex–A9, Linux, RTOS, virtio, rpmsg, remoteproc.

Keywords

AMP, SMP, SoC, FPGA, ARM Cortex–A9, Linux, RTOS, virtio, rpmsg, remoteproc.

Citace

Michal Riša: Asymetrický multiprocesing na ARM Cortex-A9, diplomová práce, Brno, FIT VUT v Brně, 2015

Asymetrický multiprocesing na ARM Cortex – A9

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Pavla Korčeka. Další informace mi poskytl Ing. Jan Viktorin. Prohlašuji, že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Riša
July 29, 2015

Poděkování

Ďakujem Ing. Pavlovi Korčekovi a Ing. Jánovi Viktorinovi za odbornú pomoc, spätnú väzbu pri písaní textu diplomovej práce, podporu a nádherný ľudský prístup. Ďakujem Ing. Jurajovi Rišovi, rodine a Lenke Barošovej za bezhraničnú podporu, ochotu pomôcť a motiváciu. Ďakujem Bc. Martinovi Rišovi za spätnú väzbu pri písaní textu diplomovej práce.

Contents

List of Figures

Chapter 1

Introduction

Today's computer systems are required to be fast, safe, cheap and energy efficient. There are a lot of ways how to achieve this. One of them is *asymmetric multiprocessing* (*AMP*).

AMP systems typically contain several homogeneous or heterogeneous processing cores (for example devices from Texas Instruments OMAP *SoC* [15] family contain ARM processor, *GPU* and *DSP*). System load is distributed among the cores forming functionality of entire system. The cores usually run different software including Linux, bare–metal or RTOS. *Symmetric multiprocessing* (*SMP*) systems balance computer system load among homogeneous processing cores.

Heterogeneous *AMP* systems benefit from different processing cores. This diversity makes it possible to accelerate some tasks on a specialized processing core. Hardware multimedia codec support allows the system to be more energy–efficient and faster than software multimedia codec. Implementing control mechanisms to real–time applications can break real–time response. One of the solutions is splitting control mechanism and the real–time application to different processing cores. Control processing core runs Linux and uses existing networking and management tools while worker core can run a RTOS for time critical tasks. One way of increasing computer system security is to separate secure and non–secure applications to different processors. Insecure applications run on processor core with reduced permissions and separated memory.

New information was discovered since creation of corresponding term project. Because of that, this thesis is rework and extension of the term project.

Chapter (2) contains brief introduction to *AMP* systems and some available platforms. Chapter (3) provides short description of the ARM Cortex–A9 processor and ARMv7–a architecture. Altera Cyclone V *SoC FPGA* is described with focus to processor subsystem in chapter (4). Proposed application, development plan and implementation is summarized in chapter (5). Thesis results and possible future work is in chapter (6).

Chapter 2

Asymmetric multiprocessing

Information in this chapter can be found in an OpenAMP documentation [16]. Other information sources are mentioned below.

According to chapter (1), *asymmetric multiprocessing* (*AMP*) is an approach to computer system load distribution among heterogeneous software or hardware environments. *AMP* system is formed by processing cores running subtasks. Because of that, communication among the processing cores is important.

Altera Cyclone V [9], Xilinx Zynq [21] and OMAP platform [15] are examples of heterogeneous *SoC* hardware devices capable of forming *AMP* system.

Heterogeneous SoCs

Altera Cyclone V and Xilinx Zynq contain ARM Cortex–A9 processors and a *FPGA*. This combination allows developers to partition the application between general–purpose single–core or multi–core ARM processors and the *FPGA*. The Altera Cyclone V device is described in chapter (4). Following figure shows simplified block diagram of the Altera Cyclone V and Xilinx Zynq *SoC FPGA*s.

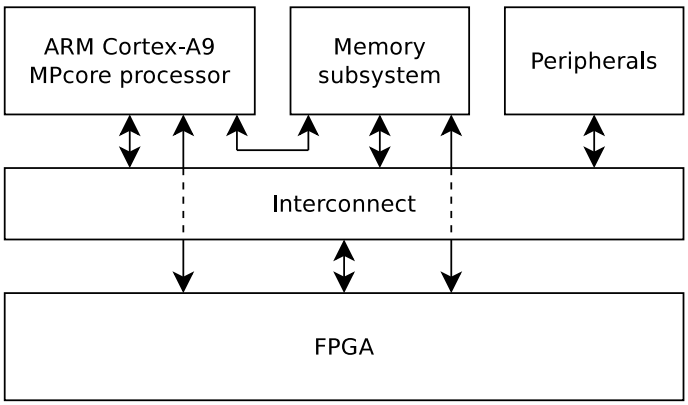


Figure 2.1: Altera Cyclone V and Xilinx Zynq *SoC FPGA* structure.

Texas Instruments OMAP 3 platform is a combination of general–purpose ARM processor and coprocessors including DSP and GPU. OMAP3530 device contains:

- Up to 720 MHz ARM Cortex–A8 processor with NEON SIMD coprocessor and *MMU*.
- Up to 520 MHz TMS320C64x VLIW DSP core.
- PowerVR SGX graphics accelerator with OpenGL ES 2.0 and OpenVG 1.0 support.

The OMAP 3 platform is designed to provide video, image and graphics support for video streaming, video conferencing and high–resolution still images. The platform is able to run Linux, Windows CE and Android operating system. Detailed information is available in datasheet [27].

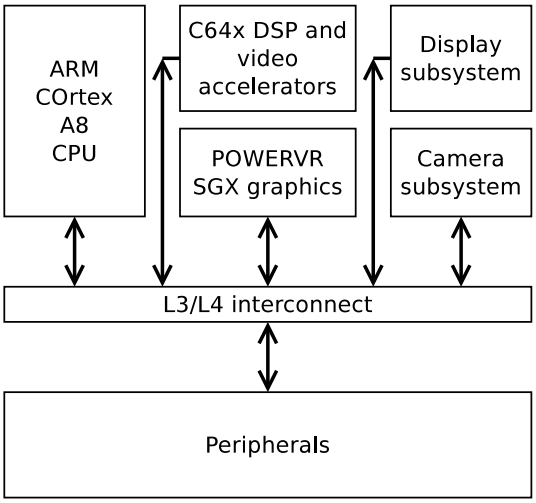


Figure 2.2: OMAP 3530 structure.

AMP system organization

AMP system contains multiple processor units that share or cooperatively use hardware resources. Following things are important in an *AMP* system:

- Memory organization.
- Boot sequence.
- Peripheral access.
- System security.
- Software being run.

2.1 Memory organization

Memory is vital system resource in a computer system. Address map, alignment and types of available memories are important aspects of *AMP* system. The memory is typically partitioned to shared and private memory regions. Following figure shows example memory partitioning.

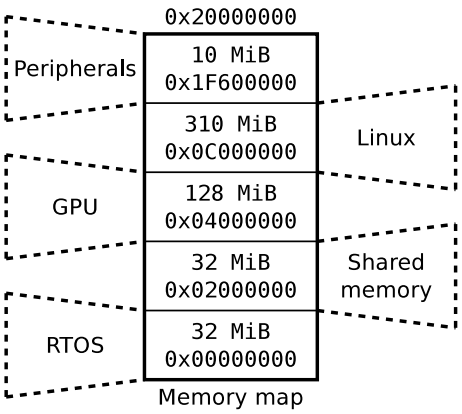


Figure 2.3: Example memory partitioning.

Several address space separation techniques exist. ARM *memory protection unit* (*MPU*) (see section 3.4) provides separation of physical address spaces by defining physical memory regions and access permissions. Another technique is to use a virtual memory. *Virtual address* (*VA*) is an address at which item seems to reside from application's point of view. *Physical address* (*PA*) is a hardware address at which the item actually resides. The virtual address needs to be translated to a physical address before the memory is accessed, thus translation process is required. *Memory management unit* (*MMU*) is a hardware unit that performs this translation.

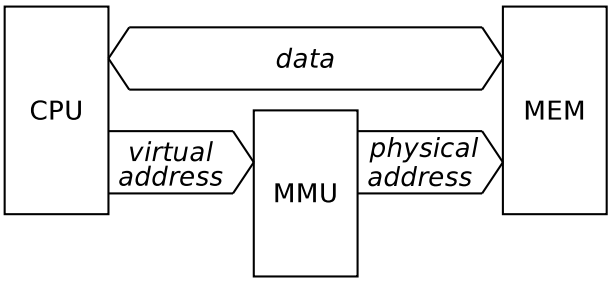


Figure 2.4: Memory management unit and virtual and physical address.

The *MMU* is used to:

- Perform memory access and permission checks. Typically, read, write and execute permissions can be set for each memory mapping.
- Isolate virtual address spaces. Each processing core maps only it's own virtual addresses to distinct physical memory regions. Access to any other than mapped virtual address causes memory access violation.
- Create shared memory segments by mapping regions of multiple virtual address spaces to the same physical memory region.
- Relocate code that needs to be placed at specific address.
- Allocate fragmented memory regions. The regions can be allocated without need to be contiguous in physical memory.

2.2 Boot sequence

Booting is a hardware–specific initialization process of a computer system. The computer system enters initial state after power on event by a hard/cold reset operation. A soft/warm reset operation does the same except it takes the system from running state to initial state. It is possible to skip *power–on self–test* (*POST*) [6] after the warm reset. A boot loader is a software that performs initial stages of the computer system initialization.

Multiple stage boot loaders exist because boot loader storage memory is typically small (about 64 KiB). First stage boot loader initializes basic hardware components and executes second stage boot loader. This way it is possible to remove the size constraint and use more powerful boot loader. Some boot loaders are shown below:

- **BIOS** [5] is a boot loader used by an IBM compatible computers. The *BIOS* boot process contains a *power–on self–test* phase. The *POST* is used in several embedded systems as well.
- **Das U–boot** [20] is an universal boot loader for embedded devices. It boots embedded device's kernel [10]. The U–boot boot loader is used in *AMP* application in this thesis.

Processor startup

A processor starts executing code at address specified by a reset interrupt vector. This address points to some sort of boot loader. Single processor computer system starts execution of the boot loader by completion of a reset operation. Multi–core computer system typically resets all processor cores but completes only single one's reset. Initialized processor executes the boot loader and is free to setup reset interrupt vectors of other processors and complete their's reset operation. Multiple *AMP* system processing cores can boot at the same time.

2.3 Peripheral access

Two basic peripheral access strategies besides turning the peripheral off exist:

- Reserve peripherals for a specific processing core.
- Cooperatively use peripherals by multiple processing cores. A mechanism of determining who and when owns the peripheral is needed.

A *direct memory access* (*DMA*) capable peripheral is potential security risk. The DMA engine operates with physical addresses. Buggy or malicious code can use the DMA engine to access memory without *MMU* permission checking. An *IOMMU* provides similar functionality as the *MMU*. It adds virtual address space and permission checking for peripheral DMA access. This avoids this risk and also provides interrupt remapping [14]. A *device address* (*DA*) is a virtual address used by the DMA engine to address physical memory. A physical address is an address where the data transferred by the DMA engine reside in the physical memory.

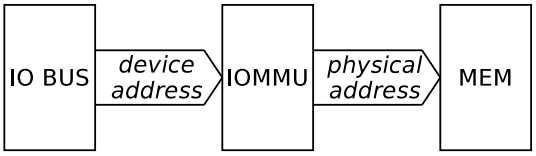


Figure 2.5: IO memory management unit (*IOMMU*).

Usage of the *IOMMU* brings the same benefits as *MMU*. *IOMMU* is able to restrict device to access specific subset of address space.

2.4 System security

AMP systems can be supervised or unsupervised [16]. Supervised *AMP* system is characterized by presence of a hypervisor that enforces isolation and cooperative resource usage. Unsupervised *AMP* system contains modified software that ensures cooperative usage of system resources. Memory address space isolation enforced by a *MMU* or *IOMMU* and usage of technologies like ARM TrustZone increases safety of unsupervised *AMP* system. This thesis is focused on unsupervised *AMP* systems.

2.5 System software

AMP system is characterized by simultaneous running of multiple operating systems. It is common to use following operating system types within single *AMP* system:

- Linux operating system.
- RTOS.
- Bare–metal firmware.

The software run within *AMP* system needs to be patched in a way that it will cooperatively use hardware resources. It is common practice for an *operating system* (*OS*) to initialize interrupt controller during startup. This is a correct behavior in a *SMP* system. But this can ruin *AMP* system because of other software run on different processor core with shared interrupt controller.

Chapter 3

ARM Cortex–A9 processor

Information in this chapter comes from ARMv7–A architecture reference manual [24], Xilinx Zynq [28] and Altera Cyclone V [23] manuals. Other information sources are mentioned below.

The ARM Cortex–A9 processor is a single core processor with full virtual memory support. The processor implements ARMv7–A architecture and runs an ARM, Thumb and Jazelle instruction sets. Typical single core Cortex–A9 design contains:

- ARM Cortex–A9 uniprocessor.
- L2 cache controller.
- Interrupt controller.

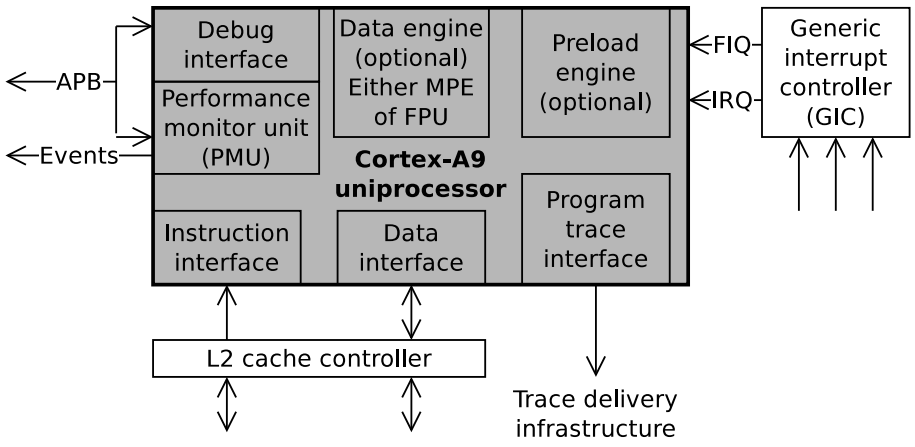


Figure 3.1: ARM Cortex–A9 uniprocessor system.

The processor can include optional media processing engine and a *FPU* (*floating–point unit*).

3.1 Variants

The Cortex–A9 processor can be used in uniprocessor and multiprocessor configurations. In the multiprocessor configuration, up to four Cortex–A9 processor cores are available in cache–coherent cluster.

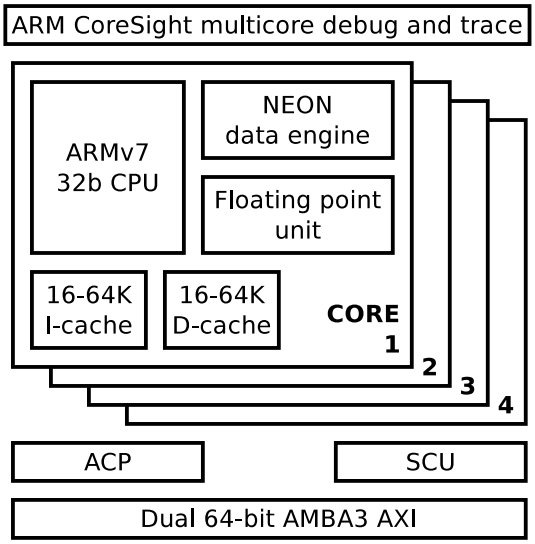


Figure 3.2: ARM Cortex–A9 MPcore processor.

The multi–core cluster contains:

- Up to four ARM Cortex–A9 processors.
- A *snoop control unit* (*SCU*) that is responsible for:
  - Coherency among L1 data caches.
  - *Accelerator coherency port* (*ACP*) operations.
  - Uniprocessor accesses to private memory regions.
- A *generic interrupt controller* (*GIC*).
- Private timer and private watchdog.
- A global timer.

3.2 Configurable options

Configurable options are implementation dependent options like amount of L1 cache. Some of the options for the ARM Cortex–A9 processor are listed below. For a full list of configurable options refer to ARM Cortex–A9 technical reference manual [8].

Option	**Value**
Instruction and data cache size	16/32/64 KiB
TLB entries	64…512
BTAC entries	512…4096
GHB descriptors	1024…16384
Instruction μTLB entries	36/64
NEON FPU	Included or not

Table 3.1: Some of the ARM Cortex–A9 configurable options.

3.3 Programmer's model

The processor implements ARMv7–A architecture. This section in brief summarizes some of ARMv7–A architecture features. For a detailed description refer to the ARM Architecture reference manual [24].

Processor operating mode and state determine:

- **Set of registers** available to the processor.
- **Privilege level** of software being executed.
- **Instruction set** being used. One of ARM, Thumb, Jazelle and ThumbEE.
- **Security state** that is either secure or normal. Some system resources are accessible only from secure state. Implementation with no security extensions provides only secure state.
- **Debug state** determines whether the processor can be halted for debug purposes.

Privilege levels dictate accessible architecture resources. The processor supports following privilege levels:

- **Unprivileged** (PL0) level. Software running at this level has limited access to the architecture resources and capabilities.
- **Privileged** (PL1) level. The processor permits access to all features of the architecture except virtualization functionality.
- **Virtualization** (PL2) level. Software running at PL2 can perform all operations available at PL1 level plus virtualization functionality.

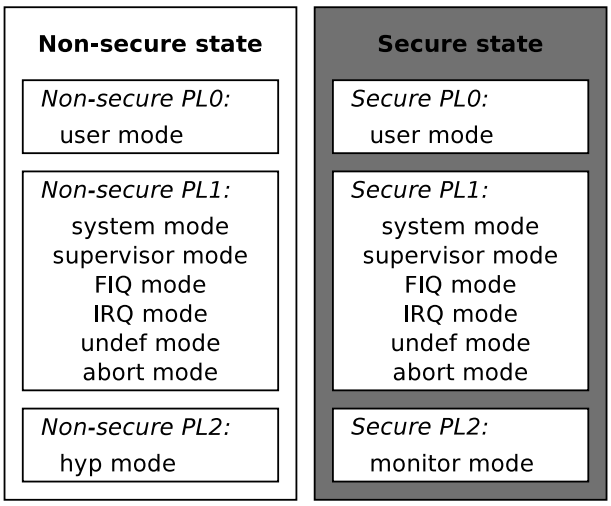


Figure 3.3: Processor operating modes and privilege levels.

Following processor operating modes are supported:

- **User** mode for unprivileged execution.
- **FIQ, IRQ** modes for FIQ and IRQ interrupt servicing mode.
- **Supervisor** mode is default mode to which a supervisor call exception is taken.
- **Monitor** mode is secure mode used to switch between secure world and normal world within ARM TrustZone security extension.
- **Abort** mode is a mode for servicing data and prefetch abort exceptions.
- **Hyp** mode is a virtualization mode.
- **Undefined** mode is default mode to which an instruction–related exception, including any attempt to execute an undefined instruction, is taken.
- **System** mode has the same registers available as user mode. This mode uses PL1 privilege level.

3.3.1 Exceptions

An exception is an event that causes the processor to interrupt normal program flow. The exception is handled by an exception handler. The handler's start address is stored in an interrupt/exception vector. The processor supports following exceptions:

- Reset.
- Interrupts.
- Memory system aborts.
- Undefined instructions.
- *Supervisor call*s (*SVC*).
- *Secure monitor call*s (*SMC*).
- *Hypervisor call*s (*HVC*).

Most of exception handling mechanisms is hidden from application developer. However, following details are visible:

- `SVC` instruction causes a supervisor call exception. This provides mechanism for unprivileged software to call an operating system or privileged software.
- `SMC` instruction causes secure monitor call exception if executed at PL1 or at higher privilege mode.
- `HVC` instruction causes a hypervisor call exception if executed at PL1 or at higher privilege level.
- `WFI` instruction provides a hint that nothing needs to be done until interrupt or similar action. This allows the processor to enter low–power state.

- `WFE` instruction does the same as `WFI` instruction plus it waits also for `SEV` instruction generated event.
- Floating–point exceptions.
- Jazelle exceptions.
- Debug events.
- ThumbEE checks events.

Wait for event (*WFE*) mechanism permits a processor in a multiprocessor system to request entry to low–power state and remain in that state until it receives an event generated by a *send event* (*SEV*) operation. *Wait for interrupt* (*WFI*) operation is also available.

3.3.2 Memory model

The processor views memory as a linear collection of bytes addressed from zero in ascending order. Instructions are always treated as little–endian. Following memory orderings are supported:

- **Normal** ordering is used for RAM and ROM devices. Code executed by the processor must be in normal memory region. The processor can do following actions on normal memory regions:
  - Repeat read and write accesses.
  - Prefetch or speculatively access memory locations.
  - Perform unaligned memory access.
  - Merge multiple accesses.
- **Device** and **strongly–ordered** orderings are more strict when compared to normal memory ordering. These orderings are used for memory mapped peripherals. These memory access rules are followed:
  - Number and size of accesses are preserved. Accesses are atomic and uninterruptible.
  - No speculative accesses and no caching is allowed.
  - Accesses can't be unaligned.
  - Order of accesses arriving at device memory is the same as program order of instructions. This applies only within the same peripheral or memory block.
  - The processor can reorder normal accesses around strongly ordered or device accesses.

  Device and strongly ordered memory differs in completion:
  - A write to strongly ordered memory is allowed co complete only when it reaches destination memory.
  - A write to device memory is allowed to complete before it reaches destination memory.

Memory attributes

Memory attributes also define the ordering of accesses for regions of memory.

- Shareability defines bus topology zones within which memory accesses are kept consistent and potentially coherent. Masters may not see the same order of memory accesses outside these zones. Refer to [28] for more information.
- Cacheability attributes apply only to normal ordered memory. This attribute controls coherency with masters outside shareability domain for a memory region. Each normal ordered memory can be assigned following cacheable attributes:
  - Write–back cacheable.
  - Write–through cacheable.
  - Non–cacheable.

Memory barriers

The ARM Cortex–A9 processor supports following memory barriers:

- ***Data memory barrier*** (*DMB*) is triggered by `DMB` instruction and ensures that all memory accesses caused by instructions before the barrier are completed before memory accesses of instructions after the barrier.
- ***Data synchronization barrier*** (*DSB*) has the same effect as DMB plus it synchronizes the memory accesses with the full instruction stream, not just other memory accesses. Execution stalls after the `DSB` instruction until all pending explicit memory accesses have completed. This does not affect instruction prefetch.
- ***Instruction synchronization barrier*** (*ISB*) flushes the pipeline and prefetch buffers. All next instructions are fetched from cache or memory. This ensures that changes in processor configuration before `ISB` instruction are visible to all current and future instructions.

3.4 Memory system

ARMv7 architecture provides two implementations of memory system:

- **Virtual memory system** that uses *memory management unit* (*MMU*).
- **Protected memory system** that uses *memory protection unit* (*MPU*).

Both of the implementations provide mechanisms to split memory to regions. Each region has specific memory types and attributes. The implementations differ in following main areas:

- MPU does not use translation tables. It uses configuration registers only.
- MPU uses only physical addresses.
- *MMU* is more complex than MPU. MPU is able to provide deterministic memory access.

MMU

Key feature of the *MMU* is address translation. It translates software virtual addresses of code and data to hardware physical addresses. This enables software to have no knowledge about physical memory map and about other software that may run at the same time. The *MMU* allows virtual address space separation and sharing.

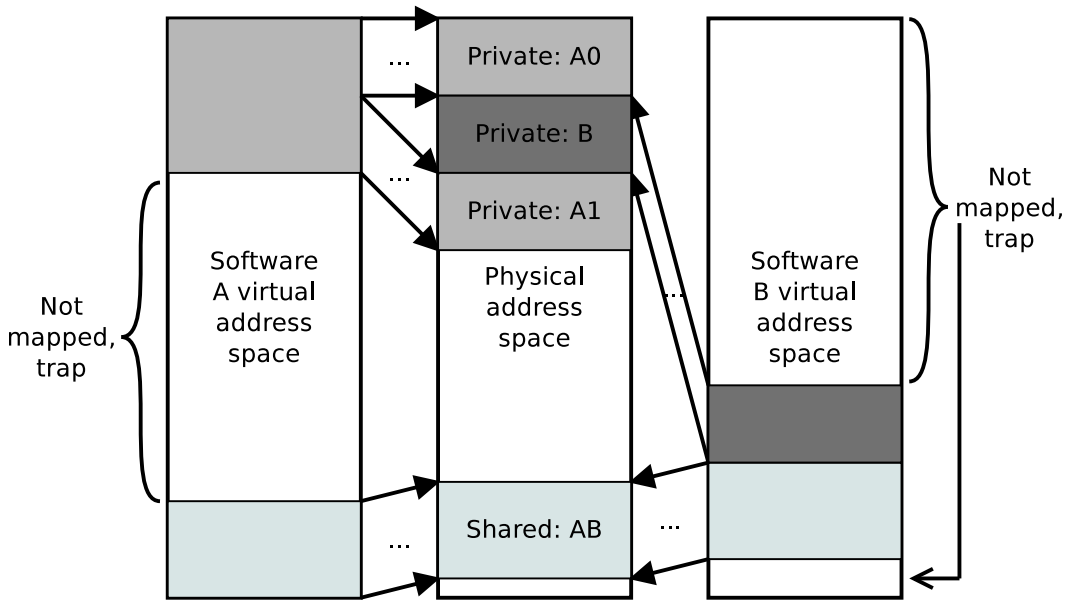


Figure 3.4: Memory address space isolation and sharing using *MMU*.

The translation process is based on translation tables containing translation entries. The *MMU* contains two major functional blocks:

- A table walker is hardware unit that automatically retrieves the correct translation table entry for a requested translation.
- A *translation look–aside buffer* (*TLB*) that caches recently used translation entries.

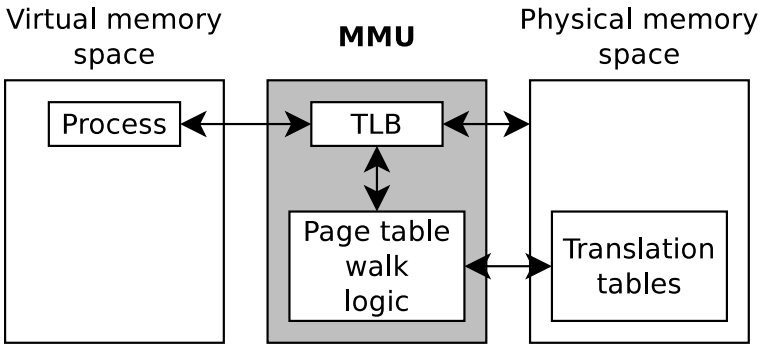


Figure 3.5: MMU functional blocks and it's surroundings.

ARMv7–A *MMU* features

- Page table entries that support 4 KiB, 64 KiB, 1 MiB and 16 MiB page sizes.
- 16 domains. A domain is a collection of memory regions. This way it is possible to group memory regions to domains.
- Global and address space identifiers. This removes need for context switch TLB flush.
- Extended permissions checking.
- Hardware page table walk.

The *MMU* uses two level TLB structure. TLB entries can be global or assigned to a process using *address space identifier*s (*ASID*). ASIDs allow TLB entries to be persistent during context switches. TLB maintenance and configuration is done through CP15 coprocessor. The TLB is organized to μTLB and main TLB.

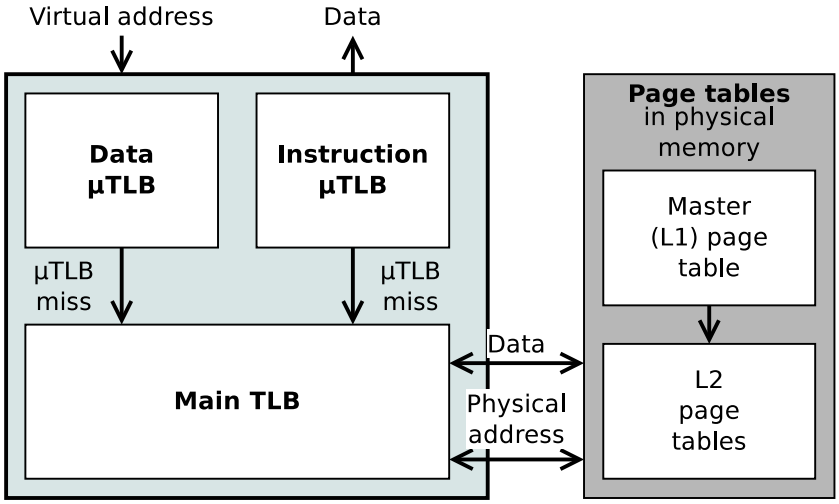


Figure 3.6: TLB organization.

- *Level 1* (*master*) page table that divides full 4 GiB address space to 4096 sections, each of size 1 MiB. There are following L1 page table entry types:
  - **Fault** that generates prefetch or data abort exception. Exception type depends on type of memory access. Fault entry indicates usage of unmapped virtual address.
  - **1 MiB section** translation.
  - **16 MiB supersection** translation (special kind of 1 MiB entry, occupies 16 1 MiB entries in a row).
  - **L2 translation table base address**.

  Base address of L1 translation table (TTB) is known and is accessible through CP15 coprocessor.
- Level 2 translation table that divides 1 MiB to:
  - 64 KiB page.

- 4 KiB page.
- A fault entry that generates abort exception like fault in L1 translation table.

μTLB

μTLB provides fully associative lookup of the virtual address within single clock cycle. The μTLB returns physical address to the cache. All main TLB operations flush instruction and data μTLBs.

Main TLB

Main TLB handles cache misses from μTLBs. Main TLB access takes variable number of cycles.

3.5 Coprocessor

A coprocessor is a non–intrusive way of extending an instruction set. Peripheral devices are usually attached to the processor by mapping it's physical registers into the coprocessor space [1]. Coprocessor instructions provide access to sixteen coprocessors described as `CP0` to `CP15`. Following coprocessors are reserved by ARM:

- `CP15` provides system control functionality:
  - Feature identification.
  - Control and status information.
  - System configuration including virtual/protected memory system configuration and performance monitor.
- `CP14` supports following areas:
  - Debug registers.
  - Thumb execution environment.
  - Jazelle direct bytecode execution support.
- `CP10` and `CP11` support floating–point and vector operations and control and configuration of the FPU and SIMD architecture extensions.
- `CP8`, `CP9`, `CP12`, and `CP13` are reserved for future use.
- `CP0` to `CP7` can provide vendor–specific features.

Most CP14 and CP15 functions can't be accessed by software running at PL0 level.

3.6 ARM TrustZone

ARM TrustZone [18] is a system–wide approach to security. It divides computer system to secure and normal world. Both normal and secure world software runs at user and privileged levels. Monitor mode software is a gatekeeper controlling migration between secure and normal world modes.

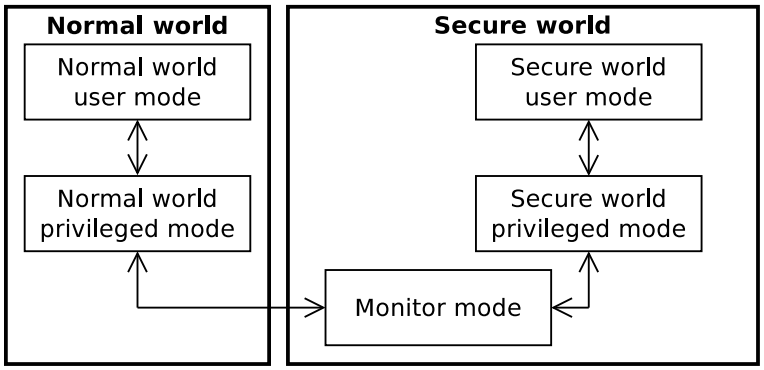


Figure 3.7: ARM TrustZone worlds and processor modes.

Processor transitions to monitor mode from the normal mode (`SMC` instruction, etc.) are tightly controlled and are viewed as exceptions to monitor software. A TrustZone–enabled processor starts in the secure world when powered on [4]. The following sequence is typical use of the security extensions:

1. Exit from reset in secure state. This happens after power on reset.
2. Configure security state of memory and peripherals.
3. Initialize the secure operating system.
4. Initialize secure monitor software to handle transitions between secure world and normal world.
5. Optionally disable modification of secure world configuration.
6. Pass control to normal world software using SMC instruction.
7. The normal world software now can initialize it's components and hardware resources it has access to.

Performance monitoring unit

The *performance monitoring unit* (*PMU*) is a per–core hardware unit for gathering statistics about operation of the processor and memory subsystem. The PMU supports 58 events and 6 counters for real–time accumulation of events. PMU counters are accessible through the processor itself using CP14 coprocessor or from an external debugger. The events are also accessible to PTM.

MPcore timers

Each processor core has interval and watchdog timer. Watchdog timer can be configured as second interval timer. Both of the timers have following features:

- 32 counters with interrupt generation at zero reach.
- Configurable starting value.
- 8–bit prescaler to qualify clock period.
- Single–shot or auto–reload mode.

Generic interrupt controller

The *generic interrupt controller* (*GIC*) supports up to 180 interrupt sources including peripheral and *FPGA* interrupts. In case of dual–core system the GIC is shared among both cores. Each processor has 16 banked *software–generated interrupt*s (*SGI*) and 16 banked *private–peripheral interrupt*s (*PPI*). These interrupts occupy number range from 0 to 31. GIC configuration registers are memory mapped by the SCU. For list of interrupt numbers refer to [23].

Snoop control unit

The *snoop control unit* (*SCU*) manages data traffic among Cortex–A9 processors, memory system and L2 cache. The SCU is responsible for:

- Maintaining data coherency between processor cores when set to SMP mode.
- Initiation of L2 cache memory accesses.
- Arbitration between processors requesting L2 access.
- Managing ACP access with cache coherency capabilities.

Following figure shows data flow among L1 data caches and the SCU.

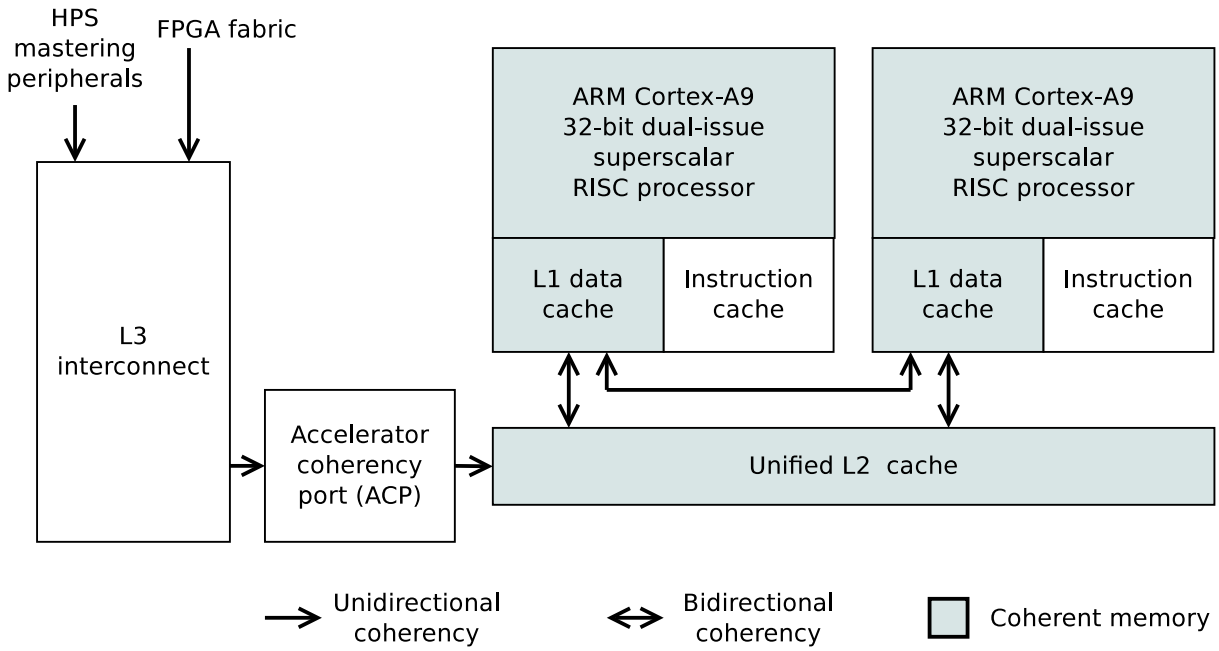


Figure 3.8: Data flow among L1 data caches and the SCU.

Chapter 4

Cyclone V *SoC* FPGA

SoC FPGA (*system–on–chip field programmable gate array*) is a device that consists of a processor system and a *FPGA*. This combination minimizes external connections between *processor system* (*HPS*) and *FPGA* and offers platform for custom hardware acceleration. The Altera Cyclone V [9] and Xilinx Zynq [21] are examples of *SoC FPGA*s.

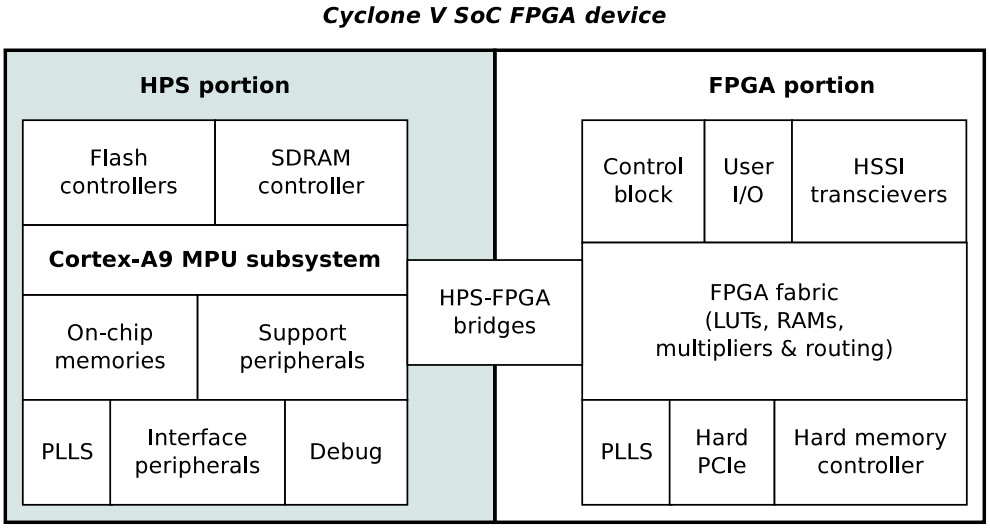


Figure 4.1: Simplified Altera Cyclone V *SoC FPGA* block diagram.

The Cyclone V consists of following components:

- A HPS that contains:
  - Support peripherals (system control, clock control, etc).
  - Interface peripherals (ethernet, USB, etc).
  - Cortex–A9 *MPU* (*microprocessor unit*) subsystem.
  - PLLs and debug circuitry.
  - Memory controllers and on–chip memory.
- A *FPGA* portion:
  - *FPGA* fabric.

- Control block.
- *Phase–locked loops* (*PLLs*).
- Depending on the variant, high–speed serial interface transceivers, hard PCI Express controllers and hard memory controllers.

- Bridges between HPS and *FPGA* portion.

The HPS and the *FPGA* portions are distinctly different. HPS is able to boot from multiple sources not excluding external flash memory and *FPGA* fabric. Both portions have their own pins and separate external power supplies. The portions can be independently power on. The HPS must be power on before or at the same time as the *FPGA* portion. Xilinx Zynq has similar architecture.

4.1 Block diagram

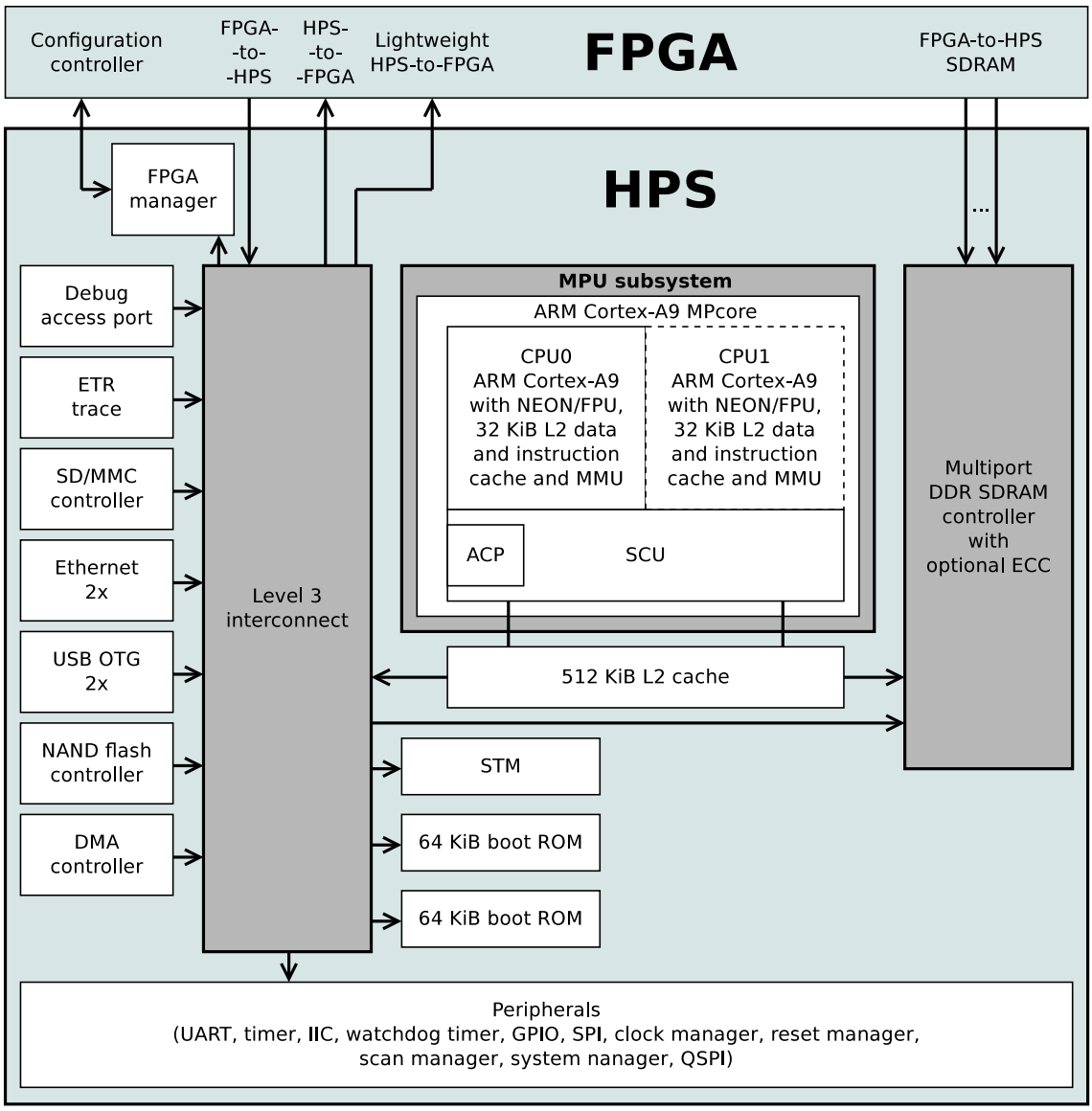


Figure 4.2: Altera Cyclone V *SoC FPGA* block diagram [23].

Short description of the Altera Cyclone V blocks follows.

Memory controllers

Following memory controllers are included:

- SDRAM controller supports DDR2, DDR3 and low–power LPDDR2 devices. ECC including single–bit error correction and write back is supported. SDRAM controller is able to address full 4 GiB address space and supports 8, 16, and 32–bit data widths. The controller provides command and data reordering with deficit round–robin arbitration with aging and high–priority bypass for latency sensitive traffic.
- NAND flash controller.
- QSPI flash controller.
- SD/MMC controller.

Support peripherals

Support peripherals provide *SoC FPGA* device control. These support peripherals are included:

- **Clock manager** allows clock settings for processor system and dynamic clock tuning. The clock manager contains these **PLL clock groups**:
  - **Main group** clocks for the Cortex–A9 MPcore processor, L3 interconnect, L4 peripheral bus and debug logic.
  - **Peripheral group** clocks for PLL clocked peripherals.
  - **SDRAM group** clocks SDRAM memory subsystem.
- **Reset manager** manages reset signals of the processor system. There are following reset domains:
  - ***Test access port*** (*TAP*) that targets JTAG TAP controller.
  - **System domain** contains all processor system except TAP domain and *FPGA* fabric connected to processor part reset signals.
  - **Debug** contains all debug logic (DAP, CoreSight, trace logic, etc.), Cortex–A9 MPcore processor and *FPGA* fabric.

  There are three **reset types**:
  - **Cold reset** (power–on reset) ensures that hardware is placed to sufficient state to boot. Resets all reset domains and all logic possible.
  - **Warm reset** occurs after processor system completed cold reset. Cold reset is used to recover Cyclone V device from non–responsive condition. This reset is a subset of cold reset and affects only system reset domain.
  - **Debug reset** takes place after cold reset and affects only debug reset domain. Debug reset is used to reset debug logic from a non–responsive condition.

- **System manager** controls these features:
  - ECC monitoring and control.
  - Pin multiplexing.
  - Low–level control for peripheral features unaccessible through `CSR` registers.
  - Freeze controller for freezing IO elements to state safe for configuration.
  - DMA engines.
  - Ethernet, processor subsystem, etc.
- **Scan manager** controls HPS I/O pins and communicates with the *FPGA* JTAG test access port controller.
- **FPGA manager** offers following features:
  - Configuration of of the *FPGA* part of the Cyclone V device.
  - *FPGA* fast passive configuration interface.
  - Partial reconfiguration.
  - Compressed *FPGA* configuration images.
  - AES encrypted *FPGA* configuration images.
- Timers.
- Watchdog timer.
- DMA controller.

Interface peripherals

Interface peripherals are communication bridge to *SoC* FPGA's surrounding environment. The *SoC FPGA* contains following interface peripherals:

- Ethernet.
- USB OTG.
- IIC.
- UARTs.
- CAN controllers.
- SPI controllers.
- GPIO.

On–chip memory

- 64 KiB on–chip RAM with ECC support.

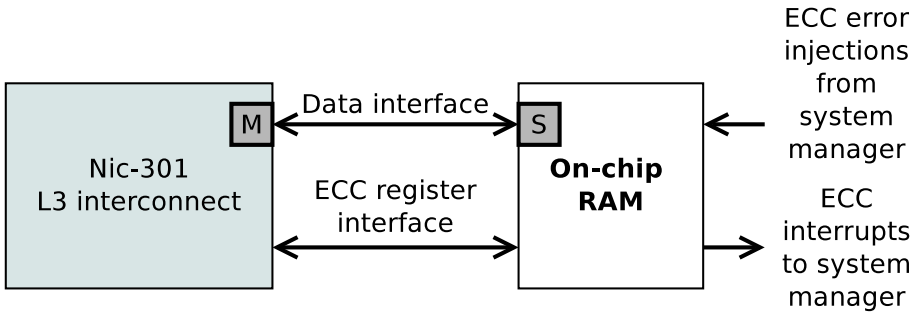


Figure 4.3: On–chip RAM block diagram.

  The on–chip ram serves as a general–purpose memory accessible from the *FPGA*. All memory accesses use strong order type. Contents of the on–chip RAM preserve reset.

- 64 KiB boot ROM with. Boot ROM contents preserve reset.

Endian support

The HPS is a little–endian system. All HPS slaves are little–endian. Processor masters are configurable to interpret data as little or big–endian. HPS–FPGA interfaces are little–endian.

Address map

There are multiple address spaces within Cyclone V device.

Name	Description	Size
MPU	MPU subsystem	4 GiB
L3	L3 interconnect	4 GiB
SDRAM	SDRAM controller subsystem	4 GiB

Table 4.1: Cyclone V device address spaces.

SDRAM address space can be accessed by FPGA–to–HPS interface from the *FPGA* fabric. Total amount of SDRAM accessible from other address spaces can be configured at runtime. The MPU address space is 4 GiB and applies to addresses generated inside the MPU. The address space contains following regions:

- SDRAM window region provides access to portion of the 4 GiB SDRAM address space.
- MPU L2 cache controller connects to L3 interconnect and to the SDRAM. Address filtering start and end registers in the L2 cache controller define the SDRAM window boundaries.

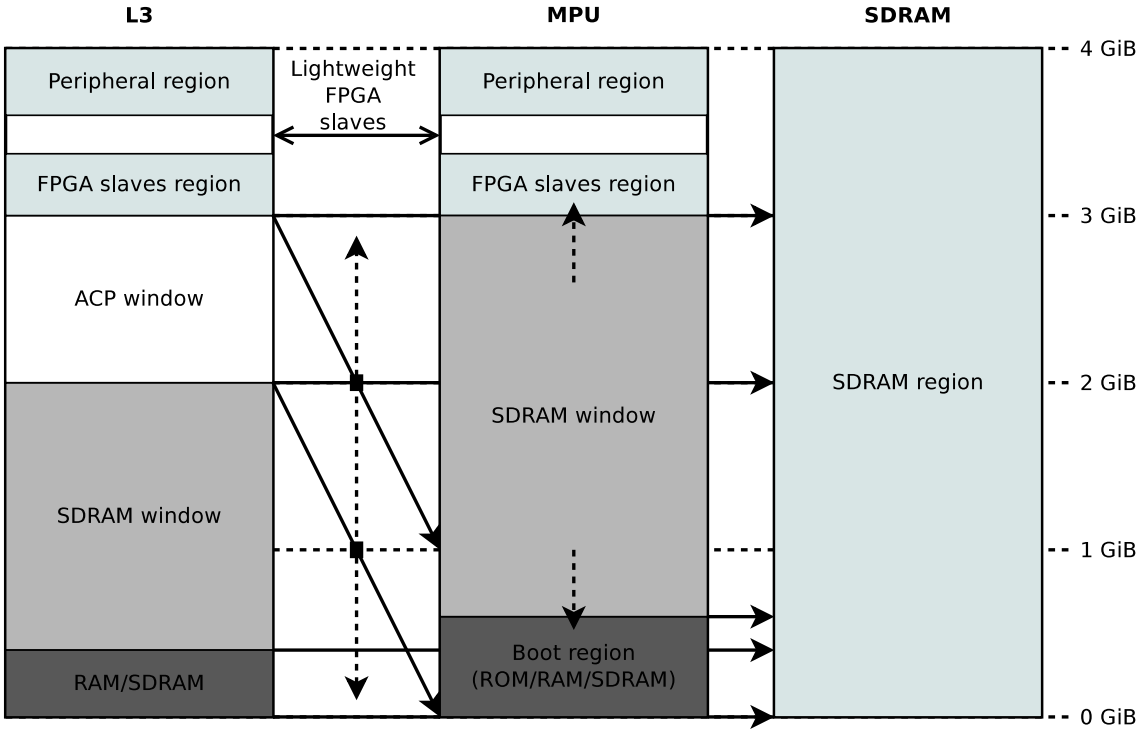


Figure 4.4: Address space relationships.

Window regions provide access to other address spaces. ACP window in L3 address space maps to 1 GiB region in the MPU address space. SDRAM window in MPU address space is able to grow or shrink modifying size of *FPGA* slaves and boot region. The ACP window can be mapped to any region in the MPU address space on a GiB aligned boundaries.

4.2 MPU subsystem

This section describes the MPU subsystem in more detail. Following figure shows the MPU subsystem with surrounding blocks.

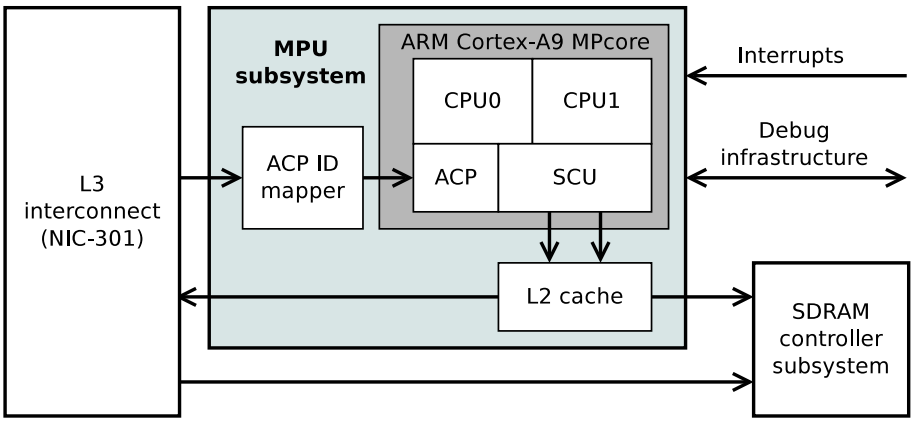


Figure 4.5: MPU subsystem block diagram.

The Cyclone V device's MPU subsystem contains:

- ARM Cortex–A9 MPCore:
  - One or two ARM Cortex–A9 processor cores in a cluster.
  - NEON SIMD coprocessor and vector FPU per processor core.
  - *Snoop control unit* (*SCU*) to maintain cache coherency within the cluster.
  - *Accelerator coherency port* (*ACP*) that accepts coherency memory access requests.
  - Interrupt controller (*GIC*)
  - One general–purpose timer and one watchdog timer per processor core.
  - Debug and trace features.
  - 32 KiB L2 instruction and data cache per processor core.
  - *Memory management unit* (*MMU*) per processor core.
- Shared 512 KiB ARM L2–310 L2 cache.
- ACP ID mapper that maps 12–bit ID from L3 interconnect to 3–bit ID of the ACP.

The ARM Cortex–A9 processor is configured with options shown if following table.

Feature	**Options**
Cortex–A9 processors	1 or 2
Instruction cache size (per core)	32 KiB
Data cache size (per core)	32 KiB
TLB size (per core)	128 entries
NEON media processing engine	Included
Preload engine (per core)	Included
Preload FIFO entries	16
Jazelle DBX extension	Full
Program trace macrocell interface	Included
Parity error detection	Included
Master ports	2
Accelerator coherency port	Included

Table 4.2: ARM Cortex–A9 MPcore configuration.

Debugging

Each Cortex–A9 processor is capable of handling six hardware breakpoints and four hardware watchpoints.

L1 caches

L1 caches are four–way set associative with 32 bytes cache line size and parity checking.

Memory management unit

The *MMU* cooperates with L1 and L2 caches to translate software virtual addresses to hardware physical addresses. Each processor core has private *MMU*.

TLB type	Memory type	Number of entries	Associativity
μTLB	Instruction	32	Full
μTLB	Data	32	Full
Main TLB	Combined	128	Two–way

Table 4.3: TLBs supported by the *MMU*.

MPU address space

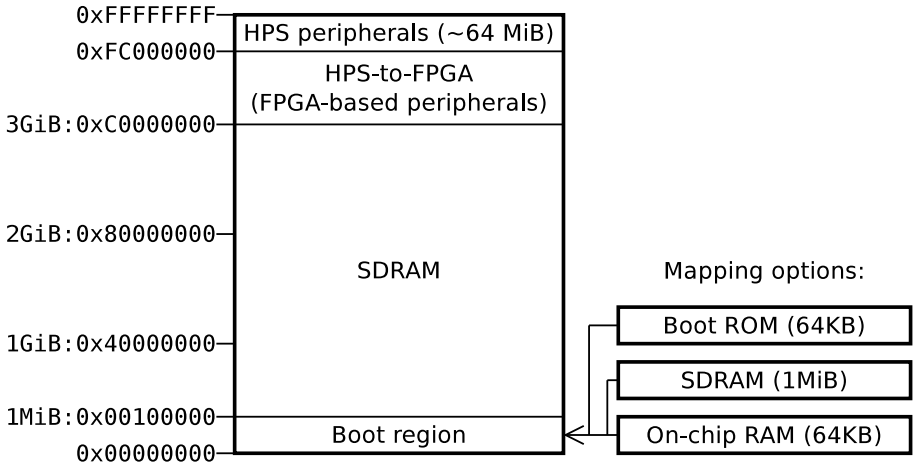


Figure 4.6: Altera Cyclone V *SoC FPGA* HPS memory map.

The MPU memory region consists of following regions:

- **Boot region** is 1 MiB in size. It's base is located at address `0x00000000`. After reset of L3 interconnect the boot region is mapped by boot ROM. This allows the processor to boot. Access above first 64 KiB if the boot region is illegal because boot ROM is 64 KiB in size.
- **SDRAM region** starts at address `0x00100000` (1 MiB). Top of the region is determined by the L2 cache filter. L2 cache address filtering defines address range of the SDRAM region. Access between start and end address of SDRAM region is routed to SDRAM. Access outside this range is routed to L3 interconnect.
- **FPGA slaves region** can be used to allow MPU to communicate with FPGA–based peripherals.
- **HPS peripherals region** is placed at top 64 MiB in the address space. This region is always allocated to the HPS dedicated peripherals of the MPU subsystem.

Accelerator coherency port

The *accelerator coherency port* (*ACP*) allows peripherals and *FPGA* fabric to be cache coherent within Cortex–A9 processors and the SCU. Entire 4 GiB address space can be accessed coherently through the ACP.

L2 cache

The MPU subsystem contains 512 KiB L2 shared unified cache. The L2 cache is eight–way associative, configurable down to one–way. L2 cache controller consists of ARM L2C–310 controller with following configuration:

- 512 KiB cache size.
- Eight–way associativity.
- Physically addressed, physically tagged.
- 32 bytes cache line size.
- Critical first world line fills.
- Support for all AXI cache modes.
- Single event upset protection including parity on TAG RAM and ECC on L2 data RAM.
- Two slave ports mastered by the SCU.
- Two 64–bit master ports connected to SDRAM controller and to L3 interconnect.
- Cache lockdown capabilities:
  - Line lockdown.
  - Lockdown by way.
  - Lockdown by AXI master.
- TrustZone support.
- Cache event monitoring.

L2 cache can access either L3 interconnect or SDRAM subsystem. The L2 cache address filtering determines how much memory is mapped to SDRAM and to HPS–to–FPGA bridge depending on the *MMU* configuration. ECC does not affect performance of the L2 cache. ECC is performed for 8–byte aligned 64 bit writes to RAM. In order to use ECC, the software is required to meet following requirements:

- L1 and L2 cache must be configured to write–back and write–allocate mode for any cacheable region.
- L3 interconnect masters write transactions must follow 8–byte alignment and 64–bit data size.

ECC corrects correctable errors and asserts correctable error signal on the AXI bus. Uncorrectable error asserts SLVERR signal in L1 memory system. Both correctable and uncorrectable signals can trigger interrupts.

Global timer

MPU contains global 64–bit auto–increasing timer typically used by an operating system. The timer is memory mapped and accessible through the SCU. Timer has following features:

- Continues to count after sending an interrupt.
- Accessible only in secure state.

Each Cortex–A9 core has private 64–bit comparator for generating a private interrupt when the global counter reaches specified value.

Boot sequence

CPU0 is released from reset automatically. If present, CPU1 is left asserted. CPU0 can de–assert CPU1 reset signal by clearing the CPU1 bit in the MPU module reset register (`MPUMODRST`). HPS boot starts when CPU0 is released from reset and executes code from boot ROM at the reset exception address. The boot process ends when the boot ROM code jumps to next stage of the boot process. The processor is able to boot from following sources:

- NAND flash memory.
- SD/MMC flash memory.
- QSPI flash memory.
- *FPGA* fabric.

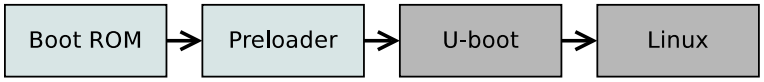


Figure 4.7: HPS boot chain example [12].

The boot ROM and preloader stages are required for the device to boot. The boot ROM content is responsible for minimal configuration of the hardware and loads preloader into 64 KiB on–chip RAM. The preloader configures clocking, pin multiplexing, DDRAM and loads next stage boot loader into RAM (U–boot in this case). Following stages are application specific. The U–boot configures the FPGA, loads Linux kernel into RAM and passes program flow to the kernel.

Chapter 5

Proposed application

AMP systems with dynamic firmware loading define master and remote processor roles. Master processor is a processor that brings up remote processor.

The proposed application consists of following components:

- Master processor running Linux.
- Remote processor running bare–metal firmware.
- Communication interface among the processors.

It was discovered that *open asymmetric multiprocessing* (*OpenAMP*) framework project [17] offers functionality for managing remote processors and communication. The OpenAMP was available only for the Xilinx Zynq hardware platform and PetaLinux software environment.

Because both Altera Cyclone V and Xilinx Zynq *SoC FPGA*s contain the ARM Cortex–A9 processor, a decision to port OpenAMP to Altera Cyclone V and to linux–socfpga [13] was made.

The OpenAMP contains demonstration applications. Porting plan of the OpenAMP was constructed:

1. Port the OpenAMP from PetaLinux to xilinx–2014.4 Linux. Test the demo application functionality.
2. Port the OpenAMP demonstration application from Xilinx Zynq to Altera Cyclone V *SoC FPGA*.
3. Port Linux–specific parts of the OpenAMP from xilinx–2014.4 Linux to linux–socfpga.
4. Test OpenAMP on the Altera Cyclone V *SoC FPGA*.

5.1 OpenAMP

OpenAMP [17] is a framework that provides software components for development of applications for *AMP* systems.

Supported environments

- **Software**: PetaLinux v2013.10 .
- **Hardware**: Xilinx Zynq–7000 all programmable *SoC* ZC702 evaluation kit.

Capabilities

- Communication among software contexts present in *AMP* system using shared memory and inter–processor interrupts for notifications.
- Proxy infrastructure for transparent interface to remote contexts from Linux master.

This thesis is focused on the life cycle management and communication.

OpenAMP components

- **Remoteproc** provides life cycle management of remote processors. This involves processor boot and shutdown.
- **Rpmsg** is responsible for message–based communication among processors within *AMP* system.
- **Virtio** is a transport abstraction that implements memory buffer management and notification mechanism for signaling availability of data in a data queue. The rpmsg uses the virtio for communication.
- Hardware and environment initialization libraries.

Rpmsg, remoteproc and virtio are part of the Linux kernel. This enables Linux applications to manage remote processors using remoteproc and to communicate with them using rpmsg. The OpenAMP can be used with *RTOS*, bare–metal firmware and Linux, as shown in figure (5.1).

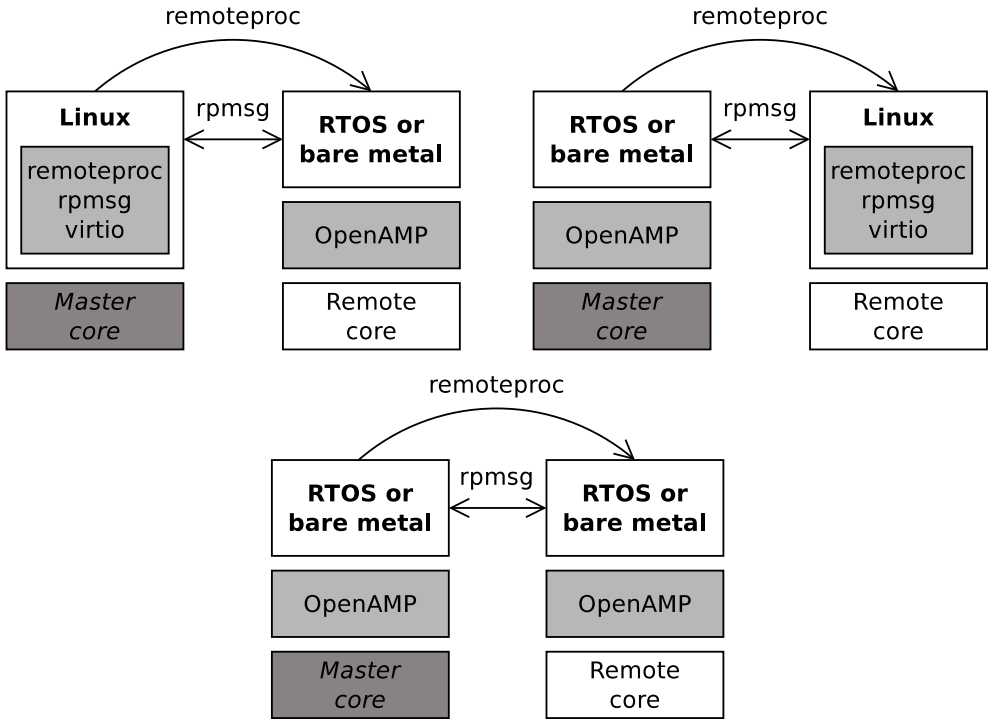


Figure 5.1: OpenAMP configurations.

Functional description

1. Master boots and uses remoteproc component to load remote firmware, allocate resources (memory and interrupts) and boot remote processor.
2. Remote boots, initializes it's resources and notifies master context that initialization is completed.
3. Master and remote do their's work and communicate.
4. Master sends shutdown request to remote context. Remote context deinitializes it's resources and halts execution.
5. Master deinitializes it's resources.
6. Master is free to shutdown entire system, continue execution or boot the same or different remote firmware.

5.1.1 Virtio

Virtio transport abstraction is an attempt to address problem of distinct virtualization systems in the Linux kernel [26].

Virtio components

- **Virtio**: A Linux–internal abstraction API.
- **Virtqueue**: A transport abstraction.
- **Virtio_ring**: A transport implementation for virtio.

Virtio

Virtio drivers register themselves to the kernel and are probed when a suitable virtio device is found. `struct virtio_device` and `struct virtio_config_ops` hold information about virtio device configuration. The configuration operations can be divided to following parts:

- Device feature bits.
- Device configuration space.
- Device status bits.
- Device reset.

These operations allow Linux to probe and configure devices and to negotiate features in forward and backward compatible manner.

Virtqueue

The virtqueue is a queue into which buffers are posted by the quest and consumed by the host. Each buffer is a scatter–gather array consisting of readable and writable generic data parts. Structure of the data is device specific. There are following virtqueue operations:

- `add_buf`: Adds new data to the queue.
- `get_buf`: Gets data from the queue.
- `kick`: Notifies a processor about queue state change.
- `disable_cb` is used to disable notification when a pending buffer is ready. It is equivalent to disabling device's interrupt. Because of expensive synchronization it is not guaranteed that notification is not sent even when disabled.
- `enable_cb` is used to enable notification about available data in the queue.

Virtio_ring

Virtio_ring is a transport implementation for virtio. It consists of three parts:

- A **descriptor table** for chaining length and address pairs.

Field	Description
`addr`	Physical address of the buffer.
`len`	Length of the buffer.
`flags`	Read/write only and next valid flags.
`next`	Number of next descriptor for chaining.

Table 5.1: `virtio_vring` descriptor table format.

  This allows a chained buffer to contain read–only and write–only buffers. By convention, read–only buffers precede write–only buffers.

- An **available ring** for indication of which descriptors are available.
- An **used ring** for indication of used descriptors.

Guest assumes all data is in it's native endianess.

Example virtio read

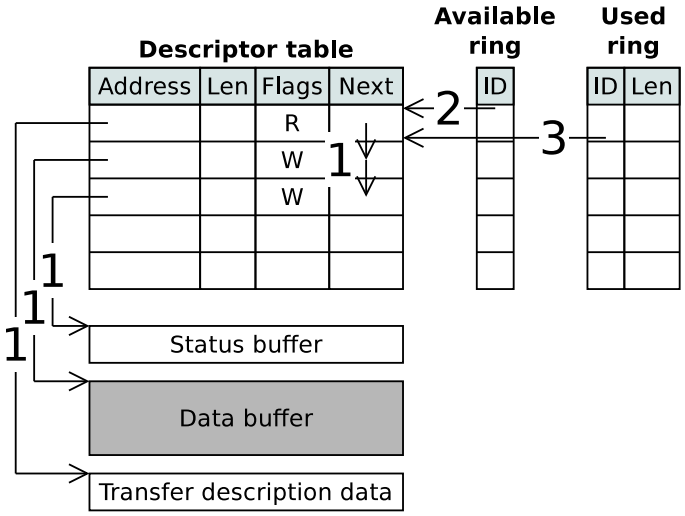


Figure 5.2: Example virtio read data structures.

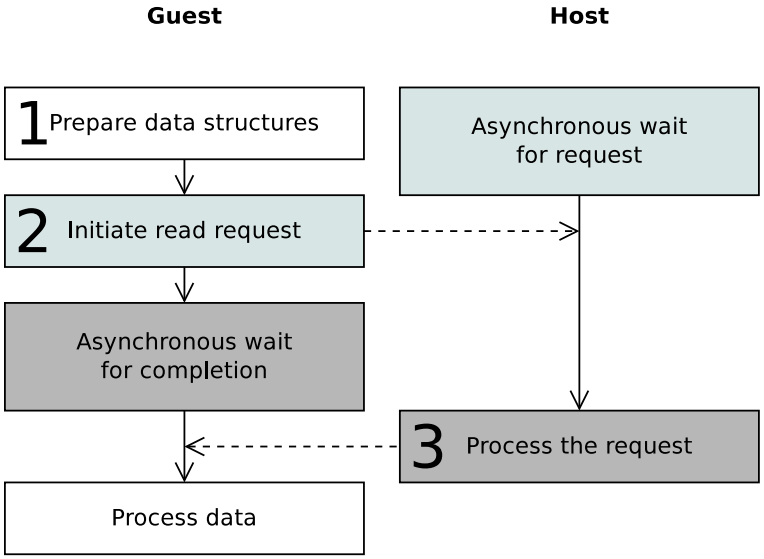


Figure 5.3: Example virtio read guest and host flow.

1. Read preparation:

   (a) The guest has an empty buffer the data will be read to (data buffer).

   (b) Transfer metadata buffer and status buffer are allocated and filled.

   (c) The data, metadata and status buffers are put into three free entries in the descriptor table and chained together. If the data buffer is not physically contiguous, multiple chained descriptor table entries are used. (1)
   The header is read–only. Status and the data buffers are write–only.

2. The descriptor head is marked available (2). This is done by placing descriptor table entry index into the available ring and issuing memory barrier. A `kick()` operation is used to notify the host that there is pending data in the queue.

3. The host completes the request at some point in the future. The data buffer is filled and status information is updated. The descriptor head is returned to the used virtqueue and the guest is notified. The guest calls `get_buf()` until `NULL` is returned.

5.1.2 Rpmsg

Rpmsg is a virtio based bus for communication with remote processors. The rpmsg uses following terms:

- **Rpmsg channel** is a representation of remote processor. The channel is used to communicate within rpmsg API. Each channel is identified by textual name.
- **Rpmsg endpoint** is an association of rpmsg channel and callback function. The endpoint has 32–bit identification address. Single channel can have multiple endpoints.
- **Local (source) address** and **destination address**. All addresses are 32–bit numbers. The local address is an address of local endpoint. Destination address is an address of remote endpoint. Endpoint addresses below 1024 are reserved for predefined services. Endpoint `RPMSG_NS_ADDR` (53) is reserved for name service announcement. Example rpmsg channel with single endpoint on each side is shown in figure (5.4). Note source and destination address values.

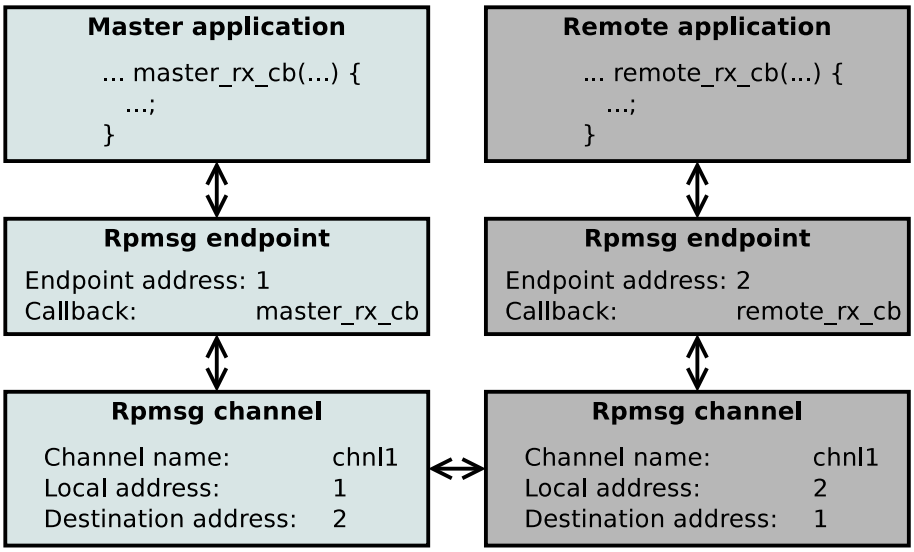


Figure 5.4: Rpmsg channel and endpoint.

Channel management

Current rpmsg implementation supports dynamic channel management. Each created channel has default endpoint. It is possible to bind multiple endpoints to single channel. Creation and deleting of a rpmsg channel causes name service announcement messages transmission.

Name service announcement

Creation and deletion of an endpoint is announced between channel participants through the `RPMSG_NS_ADDR` (53) endpoint. This endpoint is used to exchange `rpmsg_ns_msg` messages. Endpoint callback parses the message and allocates or deletes specified rpmsg channel.

Communication format

Every rpmsg message starts with a rpmsg header (`struct rpmsg_hdr`). The header contains endpoint addresses and payload information.

Field	Description
`src`	Source endpoint address.
`dst`	Destination endpoint address.
`reserved`	–
`len`	Size of data payload.
`flags`	Message flags.

Table 5.2: Rpmsg header.

Rpmsg driver implementation uses callbacks for handling message events:

- A message rx callback is called every time a rpmsg channel receives `kick()` notification. The callback does the following steps:
  1. Checks for incomming messages in channel associated virtqueue.
  2. Extracts information from a rpmsg header.
  3. Finds a rpmsg endpoint with the address equal to the address in the rpmsg header.
  4. Calls the endpoint's callback function with data payload.
  5. Returns message buffer to the virtqueue and issues the `kick()` notification to the remote processor indicating free buffer in the virtqueue.
- A message tx callback is called when the remote processor finished processing a message. This callback is used to wakeup potentially blocked rpmsg message send operations because free buffer is available.
- Name service message rx callback creates or deletes a rpmsg channel specified by the name service message content. The flags field determines whether a creation or deleting of the rpmsg channel is to be performed.

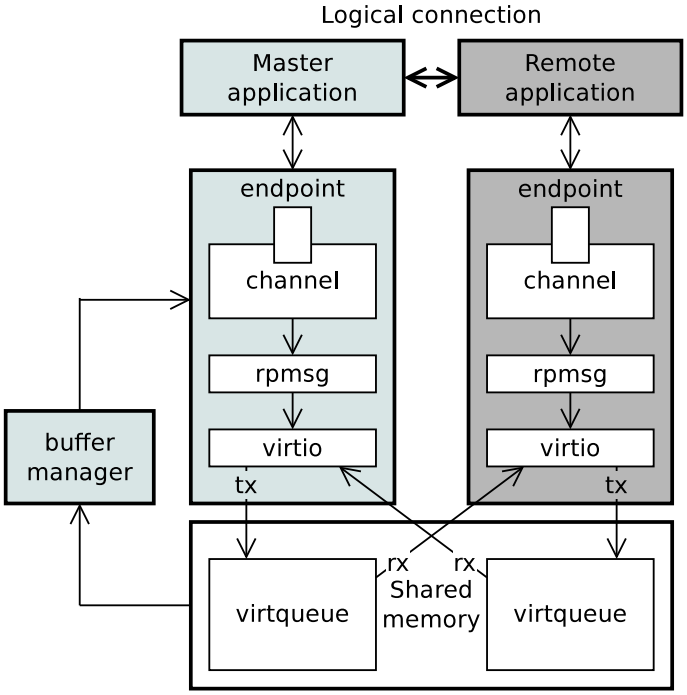


Figure 5.5: Rpmsg communication system overview.

5.1.3 Remoteproc

Remoteproc (*remote processor*) is a framework for life cycle management of remote processors in platform–independent manner. The remoteproc supports the following operations:

- Power on the remote processor.
- Boot the remote processor with a specified firmware.
- Power off the remote processor.

Resource table

A resource table is a data structure contained in remote processor firmware binary file. In case of *ELF* binary format, the table is located in the `.resource_table` section within the firmware binary file. The table is a list of system resources required and offered by the remote firmware. Resource table header format is shown in table (5.3).

Field	**description**
version	Resource table version.
num	Number of table entries.
reserved[2]	Reserved, zero.
offset[num]	Table entry offsets from beginning of the table.

Table 5.3: Resource table header format.

Table entries follow the table header. Each entry starts with an entry header containing unsigned 32–bit integer. The entry header identifies table entry type. Following table entry types are defined:

- `RSC_CARVEOUT`: information about physically contiguous memory region.
- `RSC_DEVMEM`: information about a memory–based peripheral.
- `RSC_TRACE`: announcement of a trace buffer into which the remote processor will be writing logs.
- `RSC_VDEV`: announcement of a virtio device.
- `RSC_LAST`: maximal resource entry header value. Used internally for resource entry header checking.

Example resource table is shown in figure (5.6).

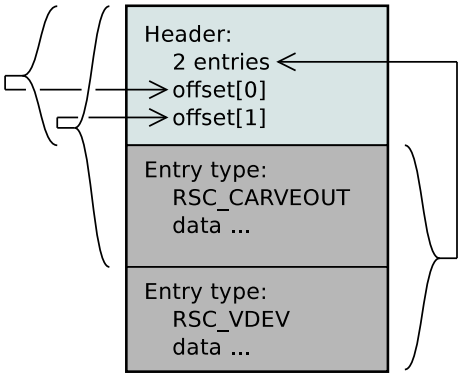


Figure 5.6: Example resource table.

Resource table parsing

The resource table is parsed before remote processor firmware is booted. Resource table parser goes through resource table entries and calls entry parsing functions according to entry types. The functions are organized in array of pointers to parser functions. It is possible to parse only specific resource table entries.

Remote processor states

The remote processor is in one of the following states:

- **Offline** state is a default remote processor state. The remote processor enters this state before remote firmware boot.
- **Running** state represents successfully booted remote processor.
- **Crashed** state is entered by an iommu fault event caused by the remote processor. The iommu fault event is caused by a violation of the iommu memory mapping settings.

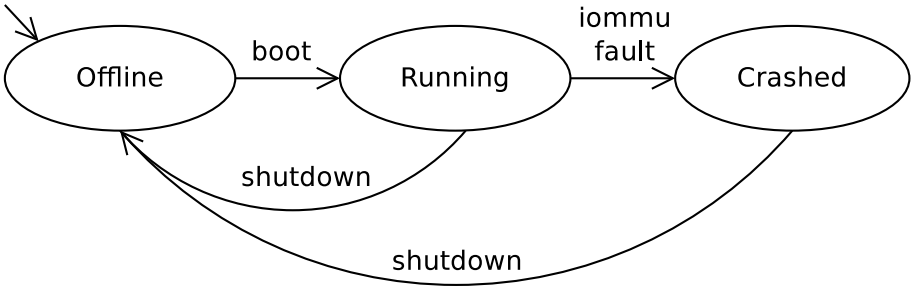


Figure 5.7: Remote processor finite state machine.

Remote processor boot

Master application boots a remote application on the remote processor in these steps:

1. Fetch and decode firmware *ELF* image.
2. Find firmware resource table and parse it's entries.
3. Allocate memory regions for remote firmware and setup necessary mappings.
4. Load code and data sections of the firmware to appropriate memory regions.
5. Release the remote processor from reset.
6. Remote processor begins execution of the firmware. Firmware initializes it's remoteproc and creates virtio and rpmsg devices necessary for communication with master processor.
7. The master receives name service announcement message. Channel created callback registered by the master application is called. The master responds to remote with a name service acknowledgement message.
8. Remote receives the message and call's it's channel created callback.
9. The rpmsg channel is established.

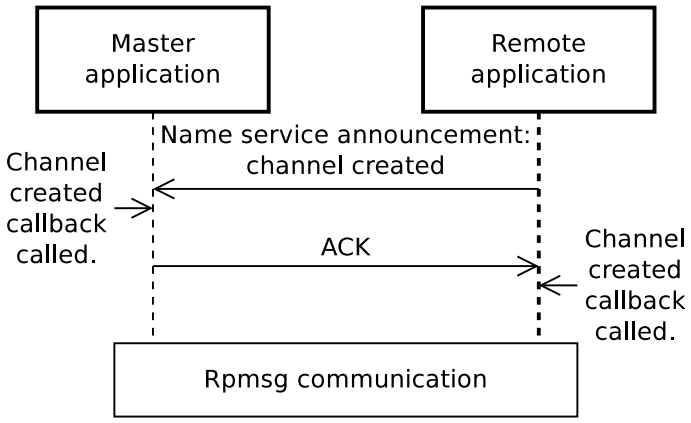


Figure 5.8: Remote processor rpmsg channel creation messages.

5.1.4 Hardware initialization library

The OpenAMP contains a hardware initialization library. The library is able to initialize:

- Stacks.
- Interrupt controller and distributor.
- Memory management unit.
- Interrupt vector table.

Macro `$PLATFORM` stands for `socfpga` for Altera Cyclone V *SoC FPGA* or `zc702evk` for Xilinx Zynq *SoC FPGA*.

MMU and caches

The *MMU* is controlled by following functions from *open-amp/libs/system/$PLATORM/baremetal/baremetal.c*:

- `arm_ar_enable_mmu`
- `arm_ar_map_mem_region`

The *MMU* requires *translation look–aside buffer* (*TLB*) address defined by `TLB_MEM_START` preprocessor symbol. Memory region mapping also sets caching options for the region being mapped.

In order to map 1MiB memory region from physical address `0xB3000000` to virtual address `0x80000000` with caches disabled, perform these steps:

1. Call `arm_ar_enable_mmu()`.
2. Call `arm_ar_map_mem_region(0x80000000, 0xB3000000, 0x100000, 1, NOCACHE)`.

The *MMU* is disabled after the processor core reset. Note that created mappings must make sense and any violation causes the processor core to enter *abort* mode.

Interrupts

The interrupt controller and distributor are controlled by the following functions from *open-amp/libs/system/$PLATORM/baremetal/baremetal.c*:

- `zc702evk_gic_initialize`
- `zc702evk_gic_interrupt_enable`
- `zc702evk_gic_interrupt_disable`
- `zc702evk_gic_pr_int_initialize`
- `arm_arch_install_isr_vector_table`
- `restore_global_interrupts`
- `disable_global_interrupts`
- `init_arm_stacks`

The interrupt controller supports eight interrupt types. Each type has it's own *interrupt vector* (*IVEC*) table entry. The *IVEC* format is shown in table (5.4).

Offset	Interrupt type
0x00	Reset
0x04	Undefined instruction
0x08	Software interrupt
0x0C	Prefetch abort
0x10	Data abort
0x14	Unused
0x18	IRQ
0x1C	FIQ

Table 5.4: Interrupt vector table format [3].

Each *IVEC* contains an *interrupt service routine* (*ISR*). The *IVEC*'s *ISR* is called for all interrupts belonging to the *IVEC*. It's up to the *ISR* to identify optional *IRQ* or *FIQ* number and to take an appropriate action. Example *IVEC* data is shown in table (5.5).

Address	Content
0x2C000000	`LDR pc, [pc, #24] ; [0x2C000020] = 0x2C000040`
0x2C000004	`LDR pc, [pc, #24] ; [0x2C000024] = 0x2C00E74C`
...	-
0x2C000020	`DCI 0x2C000040 ; Reset isr = __cs3_reset`
0x2C000024	`DCI 0x2C00E74C ; Undefined instruction ISR`

Table 5.5: Interrupt vector table content from matrix multiply firmware.

Note that *IVEC* table is followed by a table of *ISR* handler addresses. The *IVEC* table contains code that calculates correct position in handler address table for appropriate interrupt type.

In order to register a handler for a *inter–processor interrupt* (`IPI`) 8, perform following steps:

1. Setup `ARM_AR_PERIPH_BASE`, `INT_GIC_CPU_BASE` and `INT_GIC_DIST_BASE` and `ELF_START` preprocessor symbols.
2. Call `arm_arch_install_isr_vector_table(IVEC_ADDR)` with address of your *IVEC* table.
3. Call `zc702evk_gic_initialize()`.
4. Define your interrupt handler function:

```
void __attribute__ ((interrupt("IRQ"))) __cs3_isr_irq() {
    unsigned long raw_irq;
    int irq_vector;

    /* Read the Interrupt ACK register */
    raw_irq = MEM_READ32(INT_GIC_CPU_BASE +
       INT_GIC_CPU_ACK);

    /* mask interrupt to get vector */
    irq_vector = raw_irq & INT_ACK_MASK;

    if(irq_vector == 8) {
        /* process the IRQ */
    }

    /* Clear the interrupt */
    MEM_WRITE32(INT_GIC_CPU_BASE + INT_GIC_CPU_ENDINT,
       raw_irq);
}
```

Listing 5.1: IRQ handler from OpenAMP's baremetal library.

5. Call `zc702evk_gic_interrupt_enable(8, INT_TRIG_TYPE, prio)` with desired trigger type and *IRQ* priority. For list of trigger types see *open-amp/libs/system/$PLATORM/baremetal/baremetal.h*.

All interrupts are disabled after the processor core reset.

Stacks

In order to setup stacks:

1. Define preprocessor symbol `ARM_AR_ISR_STACK_SIZE`. The library creates four stacks for *IRQ*, *FIQ*, *SUP* and *SYS* processor modes.
2. Call `init_arm_stacks()`.

The `init_arm_stacks` switches the processor to mentioned modes and sets stack pointer and size to corresponding stack from point (1) of the list.

5.1.5 Marix multiply demo application

The OpenAMP contains *matrix multiply* (*matmul*) demonstration application consisting of following components:

- Kernel and user–space master applications running on CPU0 Linux master. Kernel–space parts include following *loadable kernel modules* (*LKM*s):
  - virtio.
  - virtio_ring.
  - virtio_rpmsg_bus.
  - rpmsg_mat_mul_kern_app or rpmsg_user_dev_driver.
  - remoteproc.
  - zynq_remoteproc.

  The matmul also contains a mat_mul_usr_app user–space application. Figure (5.9) shows all master matmul components.

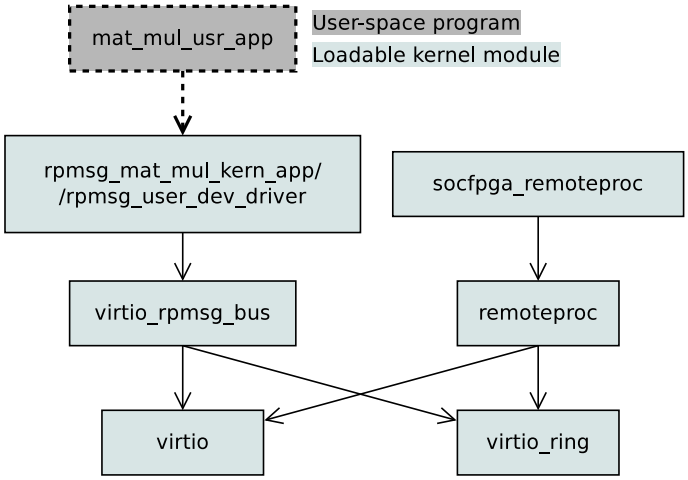


Figure 5.9: Matmul master components.

- Bare–metal remote firmware running on Cortex–A9 CPU1 remote processor.

Memory map

The matrix multiply application uses 256 MiB for RAM region and 64 MiB for ROM region. Memory region parameters are shown in table (5.6).

Region	Size	Permissions	Address
RAM	256 MiB	rwx	0x00000000
ROM	64 MiB	r-x	0xE4000000

Table 5.6: Matrix multiply memory regions.

The Linux master memory is located above the matmul firmware memory. The memory map is shown in figure (5.10).

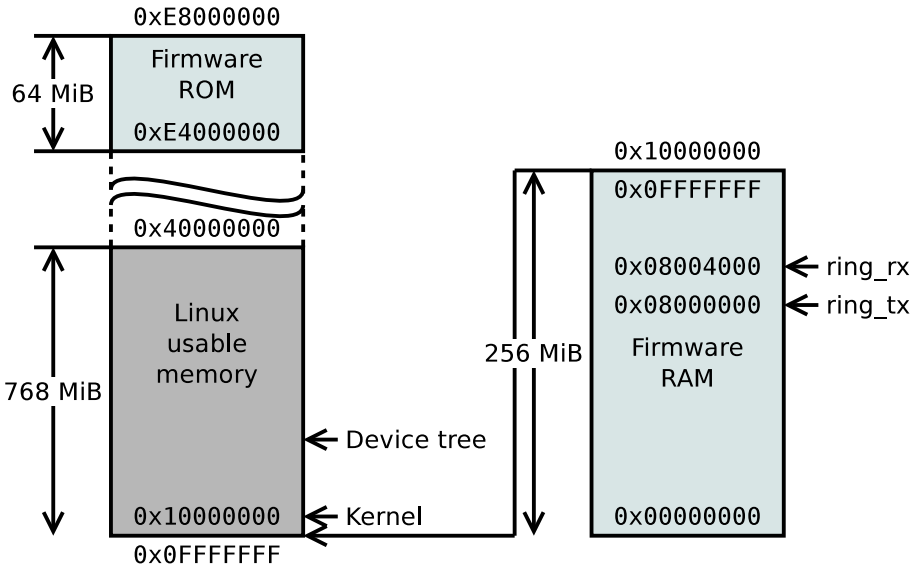


Figure 5.10: Matrix multiply application memory map.

Master kernel–space *LKM*s

- **The zynq_remoteproc *LKM*** is part of the xilinx-2014.4 Linux. The *LKM* contains:
  - Remoteproc operations for starting, stopping and notifying remote processor.
  - *Inter–processor–interrupt* (*IPI*) handler registration.
  - Remoteproc subsystem initialization.
  - Resource cleanup.

  The *LKM* performs following steps when probed:

  1. Declares DMA coherent memory for firmware.
  2. Registers *IPI* routine.
  3. Reads vring and firmware properties.
  4. Initializes remoteproc with remote processor management operations and configuration loaded from the device tree.
  5. Registers the remoteproc instance.

  Arrival of master kick notification *IPI* is deferred outside of *IPI* handler. Deferred work handler flushes caches and notifies remoteproc virtuque about received notification.

  Linux implementation of virtio, rpmsg and remoteproc handles remaining aspects of remote processor management and communication.

The zynq_remoteproc *LKM* is configured in two ways:

  - Device tree should contain entry with compatible property set to `xlnx,zynq_remoteproc`. This entry contains configuration information for the *LKM*.
  - It is possible to override some of the configuration options by modprobe command line parameters.

- **The rpmsg_user_dev_driver** does these operations:

  1. Registers a rpmsg driver for a rpmsg device named rpmsg-openamp-demo-channel. The matmul baremetal firmware contains a rpmsg device with the same name.
  2. When the rpmsg device with matching name is found, the *LKM* creates a character device `/dev/rpmsg` allowing user–space programs to access created rpmsg channel. The *LKM* supports `open`, `read`, `write` and `ioctl` operations.

- **The rpmsg_mat_mul_kern_app** *LKM* also registers a rpmsg driver for a rpmsg device named rpmsg-openamp-demo-channel. When the rpmsg device is found, the *LKM* generates two matrices, sends them to the remote, receives results and prints them.

Master user–space application

The master user–space application does following steps:

1. Opens the `/dev/rpmsg` device.
2. Queries the rpmsg device information using the `ioctl` call.
3. Creates an ui_thread and a compute_thread threads.
4. The ui_thread performs the following:
   (a) Generates random matrices.
   (b) Writes it's data to the `/dev/rpmsg` device causing the data to be sent via associated rpmsg channel.
5. The compute_thread performs these actions:
   (a) Does a blocking `read` call waiting for data from the remote.
   (b) Prints the results.
6. Sends shutdown message to the remote.
7. Closes the `/dev/rpmsg` device.

Matmul firmware

The firmware performs following steps on startup to initialize the hardware:

1. ARM Cortex–A9 CPU1 jumps to firmware code. All interrupts are disabled and the CPU1 is in after–reset state.
2. `cs3` library is initialized and then execution jumps to `main()` function.
3. CPU1 is switched to *system* mode.
4. Interrupt vector table is placed to physical address of the firmware image.
5. MMU is enabled. Mappings for firmware image, peripherals, caching options and translation table buffers are set.
6. IRQ, FIQ, SVC and SUP stacks are initialized.
7. Interrupt controller and distributor is configured and enabled.

The AMP application is set up after the hardware is initialized:

1. Resource table is parsed.
2. The firmware sets up virtio, rpmsg and remoteproc resources.
3. Interrupt handlers for virtio queues are registered.

The firmware enters infinite loop and active waits for notification from the Linux master. Figure (5.11) shows firmware flowchart.

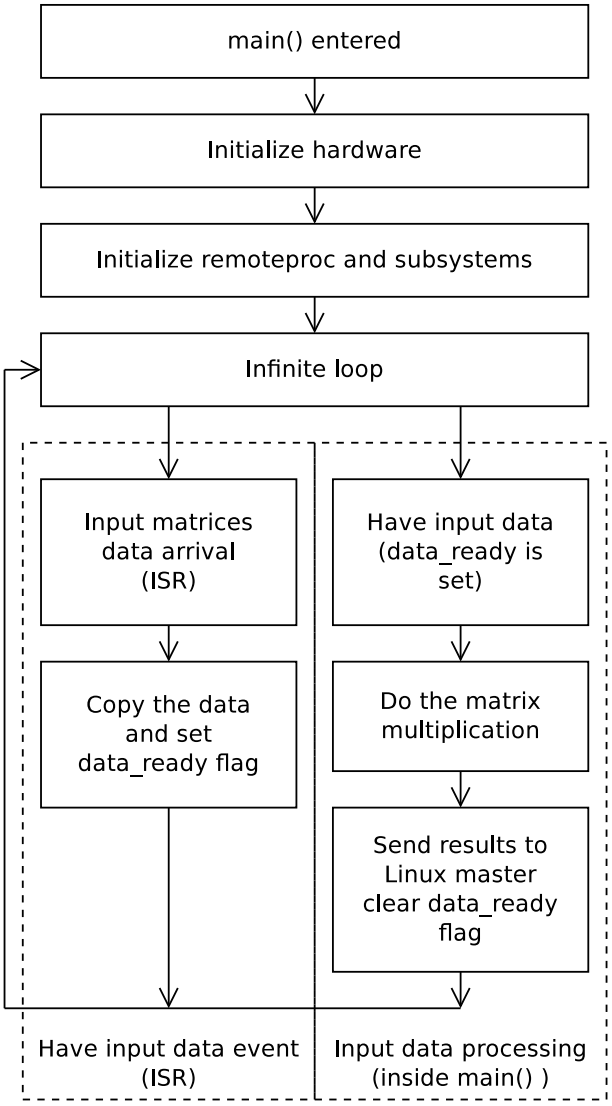


Figure 5.11: Matrix multiplication firmware flowchart.

Data is exchanged through shared memory with disabled caching. Availability of data is signaled by *IPI*s.

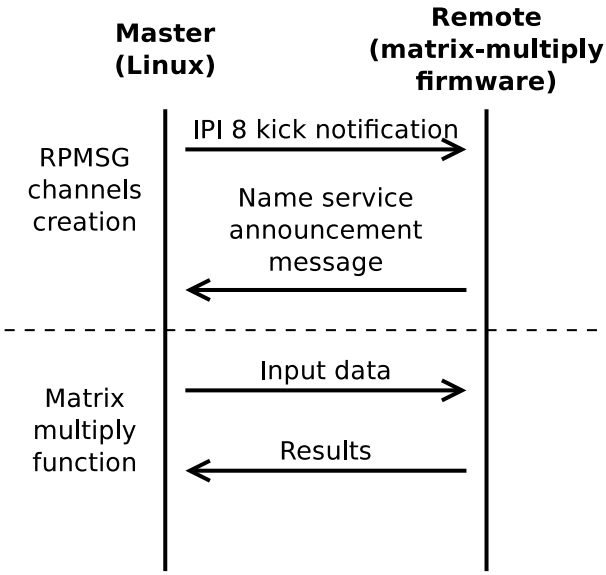


Figure 5.12: Matrix multiplication firmware communication.

Functional description

1. Linux master boots.
2. Linux remoteproc is initialized by loading zynq_remoteproc *loadable kernel module* (*LKM*).
3. Remote processor is booted by user–space application (rpmsg_user_dev_driver and mat_mul_usr_app) or kernel–space *LKM* (rpmsg_mat_mul_kern_app) on Linux master.
4. Rpmsg communication channel is established between the master and the remote processors.
5. The master generates matrices and sends them to the remote.
6. The remote receives the matrices, performs matrix multiplication and sends result back to the Linux master.
7. The master prints received result.
8. Removing of the both matmul and zynq_remoteproc *LKM*s shuts down the application.

5.2 Implementation

Implementation of the proposed application involves:

- Understanding of hardware operation.
- Linux compilation, patching and bootable card image creation.
- Bare–metal firmware compilation.
- Bootloader and kernel debugging. Debugging symbols relocation.
- Porting of the OpenAMP framework.
- Creation of *LKM* for linux-socfpga remoteproc configuration.
- Testing.

5.2.1 OpenAMP porting

Following OpenAMP porting sequence was identified:

1. Testing OpenAMP matrix multiply application with xilinx-2014.4 Linux instead of PetaLinux.
2. Porting the matrix multiply application from Xilinx Zynq to Altera Cyclone V *SoC FPGA*.
3. Porting the zynq_remoteproc *LKM* from xilinx-2014.4 Linux to linux-socfpga.
4. Testing ported OpenAMP matrix multiply application with linux-socfpga.

Running matrix multiply application with xilinx-2014.4 Linux

The xilinx Linux needs to be compiled with following options:

- `Kernel/Load address` = `0x10000000`.
- `Kernel/Device tree support` enabled.
- Use device tree source with correct `zynq_rpmsg` and `zynq_remoteproc` sections.
- `Enable loadable module support` enabled.
- `Kernel features/Memory split` = `2G/2G user/kernel split`.
- `Kernel features/High memory support` enabled.
- `Device drivers/Generic driver options/Userspace firmware loading support` enabled.
- `Device drivers/Remoteproc drivers/Support ZYNQ remoteproc` enabled.

It is also needed to modify the device tree source file [19]. Sections containing zynq_remoteproc *LKM* settings were added to the device tree. The matrix multiply application was successfully tested with xilinx-2014.4 Linux.

Porting matrix multiplication application to Altera Cyclone V *SoC* FPGA

Although Altera Cyclone V and Xilinx Zynq use the same ARM Cortex–A9 processor, memory maps of the *SoC FPGA*s differ. It was needed to change memory addresses of the Cortex–A9 subsystem and modifications were also made to OpenAMP's bare–metal library. This step was completed successfully and final firmware has been produced. Later porting has shown that it is needed to change memory layout of the matrix multiply application, because the linux-socfpga with `Kernel/Load address` set to `0x10000000` did not boot. I was unable to solve this problem. I decided to change memory map of entire AMP system solving the kernel boot problem. Modified memory map is shown in figure (5.13).

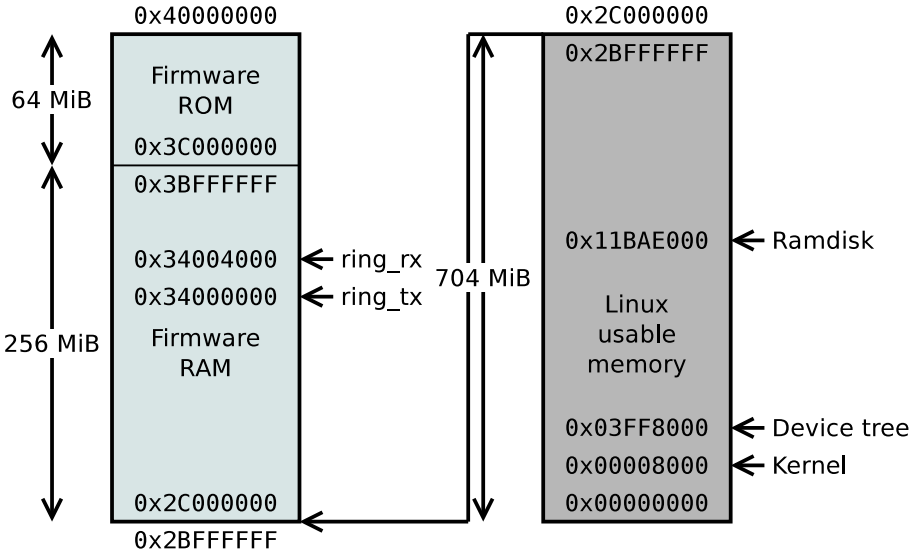


Figure 5.13: Final Altera Cyclone V memory map for proposed application.

Note that the bare–metal firmware is located above the Linux usable memory. This modification has required changes to the matrix multiply application ported to Altera Cyclone V *SoC FPGA*. The application linker script was changed according to memory region addresses in table (5.7).

Region	Size	Permissions	Address
RAM	256 MiB	`rwx`	0x2C000000
ROM	64 MiB	`r-x`	0x3C000000

Table 5.7: New matrix multiply memory regions.

The Linux is restricted from using memory reserved for bare–metal application by `mem=704m` kernel boot parameter. This implies rewrite of remoteproc memory allocation code because the socfpga_remoteproc *LKM* needs to allocate memory at specific physical address. The allocation and deallocation is done by `ioremap_nocache()` and `iounmap()`.

5.2.2 Socfpga_remoteproc *LKM*

Zynq remoteproc *LKM* for loading OpenAMP demonstration applications is not present in linux-socfpga. Socfpga remoteproc *LKM* based on the Zynq remoteproc *LKM* was created.

Following changes were made:

- Remote processor management and interrupt redirection functions are not accessible in the linux-socfpga. These functions were made visible.
- Minor changes were made to the *LKM* by renaming zynq to socfpga.
- Linux-socfpga does not allow registration of custom *IPI* handlers by default.
- Firmware memory allocation is split to firmware and buffer memory regions and *IPI* handler registration is made possible.
- Firmware image is placed at specific memory address using `ioremap_nocache`. Shared buffers for communication are allocated by `dma_declare_coherent_memory` and `dma_alloc_coherent`.

  This split is needed because when using the `dma_*` functions, allocated memory region physical address depends on order of `dma_alloc_coherent` calls. Original Linux rpmsg drivers allocate firmware image memory region after communication buffers. This leads to misplacement of firmware image and AMP system malfunction.

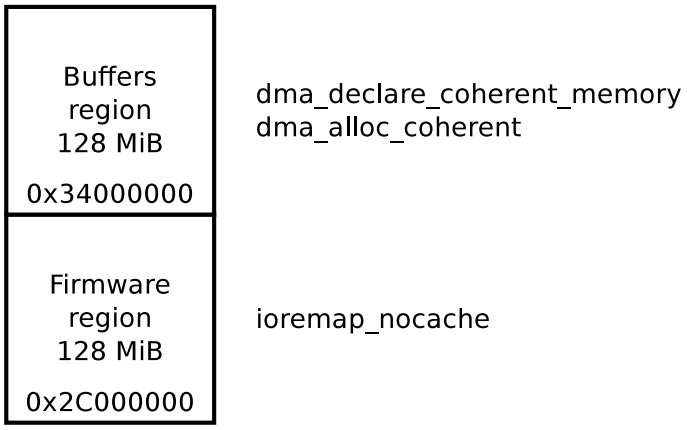


Figure 5.14: Firmware memory split to image and buffer memory regions.

Memory split required change of the device tree entry.

```
/* socfpga_remoteproc entry */
compatible = "altr,socfpga_remoteproc";

/* new reg assignment */
/*      Firmware region.      Buffers region. */
reg = < 0x2C000000 0x08000000 0x34000000 0x08000000 >;

/* original reg assignment */
/* reg = < 0x2C000000 0x10000000 >; */
```

Listing 5.2: socfpga_remoteproc *LKM* device tree entry change.

- *IPI* handler registration is added by a custom kernel patch. *IPI* 8 and 9 are redirected to socfpga_remoteproc `LKM`.

5.2.3 Other used tools, documents and software

Several other additional tools and information sources were used during the develompent process. This subsection lists the additional sources.

- Das U–Boot [20] is the Universal Boot Loader. The U–boot is used as a second stage boot loader on the Altera Cyclone V and Xilinx Zynq *SoC FPGA*s.
- Buildroot [7] is a set of makefiles for simplifying creation of embedded Linux. The Buildroot was used to generate Linux images in this thesis as well.
- Linux device drivers, third edition [25] is a summary of loadable kernel module background. This information source was used to get deeper understanding of the Linux *LKM*s.
- ARM DS-5 development studio [2] and debugging quick start document [11] were used to debug the bootloader, Linux kernel and the firmware.

5.2.4 Running matmul application

It is necessary to meet following goals in order to run the matmul application (5.1.5):

1. Patch the linux-socfpga.
2. Compile the kernel.
3. Compile the OpenAMP.
4. Compile the OpenAMP kernel–space and user–space components.
5. Copy the matmul application firmware file to a targed Linux filesystem.
6. Create a bootable SD card image.
7. Boot the Linux on the Altera Cyclone V *SoC* development kit.
8. Connect to the kit via serial console.
9. Login as root.

- To run user–space matmul application, perform following steps:
  1. `modprobe socfpga_remoteproc`
  2. `modprobe rpmsg_user_dev_driver`
  3. `./mat_mul_demo`
  4. `modprobe -r rpmsg_user_dev_driver`
  5. `modprobe -r socfpga_remoteproc`
- To run kernel–space matmul application, follow these instructions:
  1. `modprobe socfpga_remoteproc`
  2. `modprobe rpmsg_mat_mul_kern_app`
  3. `modprobe -r rpmsg_mat_mul_kern_app`
  4. `modprobe -r socfpga_remoteproc`

5.3 Testing

As the *AMP* systems are complex, it is needed to test them properly. It is very important to verify system's functionality in small steps. The proposed application testing has shown following information:

1. Successful **linux-socfpga kernel boot** was achieved by modifying system memory map. Debugging of the kernel with default system memory map has shown hidden kernel Oops because of invalid virtual memory access. Reason of the kernel Oops is unknown.

2. **Booting of the CPU1** has been problematic. Although the CPU1 jumped to correct address (**0x2C000000**), the firmware was corrupted. A change of a memory allocation in a remoteproc *LKM* solved the issue. `ioremap_nocache` places firmware binary at specific location to the RAM (`0x2C000000`).

3. The linux-socfpga did not contain API for custom *IPI* handler functions. The API was added.

4. Successful communication among the processors was achieved by a modification to `env_map_vatopa` function from *open-amp/porting/env/bm_env.c*. According to the virtio paper [26], buffer descriptors are required to contain physical addresses of the buffers. The Linux initializes the descriptors with invalid physical addresses moved by a constant offset. The modification calculates correct physical addresses.

The matmul application works on the evaluation board. Captured communication is shown in chapter (B) and application console output is in chapter (C).

Chapter 6

Conclusion

The goal of this thesis is a demonstration of working *AMP* system allowing communication among ARM Cortex–A9 processor cores. Theoretical principles and background needed to form the application are documented. Communication is demonstrated by the OpenAMP's matrix multiply application. Port of the OpenAMP framework to the Altera Cyclone V platform and to linux-socfpga3.15 is the contribution of this thesis. The porting process is documented in section (5.2.1).

The demonstration application is able to boot a custom baremetal firmware on a secondary ARM Cortex–A9 processor. The firmware and the primary processor communicate using virtio, rpmsg and remoteproc components. Besides the firmware image, the application also contains user and kernel–space modules which are documented in section (5.1.5).

As a future work I suggest deeper testing of the AMP system and minor changes to the OpenAMP and the demonstration application. Communication latency and speed measurement will show interesting data about the communication. Usage of a *direct memory access* (*DMA*) engine can be added to both Linux and the baremetal firmware.

Arrival of platforms like Xilinx Zynq Ultrascale require effective communication and processor management mechanisms. OpenAMP may provide elegant solution.

The Xilinx Zynq UltraScale [22] family contains application, real–time and graphic processing units. Single device consists of Quad ARM Cortex–A53 processor, Dual ARM Cortex–R5 processor and ARM Mali–400 MP GPU plus a *FPGA*. This makes the Xilinx UltraScale platform an interesting candidate for *AMP* systems.

Bibliography

[1] ARM architecture - Wikipedia, the free encyclopedia. [online], [cit. 24.5.2015].
URL https://en.wikipedia.org/wiki/ARM_architecture#Coprocessors

[2] ARM DS-5 Development Studio. [online], [cit. 24.7.2015].
URL http://ds.arm.com/

[3] ARM Generic Interrupt Controller HOWTO. [online], [cit. 24.7.2015].
URL http://community.cadence.com/cadence_blogs_8/b/sd/archive/2011/07/22/arm-generic-interrupt-controller-architecture-howto

[4] ARM Security Technology Building a Secure System using TrustZone Technology. [online], [cit. 18.5.2015].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch05s01s02.html

[5] BIOS - Wikipedia, the free encyclopedia. [online], [cit. 25.5.2015].
URL https://en.wikipedia.org/wiki/BIOS

[6] Booting - Wikipedia, the free encyclopedia. [online], [cit. 25.5.2015].
URL https://en.wikipedia.org/wiki/Booting

[7] Buildroot: making embedded Linux easy. [online], [cit. 24.5.2015].
URL http://buildroot.uclibc.org/

[8] Cortex-A9 Technical Reference Manual. [online], [cit. 19.5.2015].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407i/index.html

[9] Cyclone V - Overview. [online], [cit. 18.5.2015].
URL http://www.altera.com/literature/hb/cyclone-v/cv_51001.pdf

[10] Das U-boot - Wikipedia, the free encyclopedia. [online], [cit. 25.5.2015].
URL https://en.wikipedia.org/wiki/Das_U-Boot

[11] DS-5 Workshop: Linux Kernel and Application Debug, Trace and Profile on i.MX53 Quick Start Board. [online], [cit. 24.7.2015].
URL http://cdn.shopify.com/s/files/1/0111/0072/files/DS-5_Workshop-v5.9-d1304-6-DSTREAM-i.MX53.pdf?1478

[12] GSRD - Boot Flow · Documentation · RocketBoards.org. [online], [cit. 21.5.2015].
URL http://rocketboards.org/foswiki/Documentation/GSRDBootFlow

[13] Index of /linux-socfpga.git. [online], [cit. 23.5.2015].
URL http://git.rocketboards.org/linux-socfpga.git/

[14] IOMMU - AMD. [online], [cit. 25.5.2015].
URL http://developer.amd.com/community/blog/2008/09/01/iommu/

[15] OMAP - Wikipedia, the free encyclopedia. [online], [cit. 17.1.2015].
URL https://en.wikipedia.org/wiki/OMAP

[16] OpenAMP Framework User Reference. [online], [cit. 18.5.2015].
URL
https://github.com/OpenAMP/open-amp/raw/master/docs/openamp_ref.pdf

[17] OpenAMP/open-amp · GitHub. [online], [cit. 24.5.2015].
URL https://github.com/OpenAMP/open-amp

[18] TrustZone - ARM. [online], [cit. 17.1.2015].
URL
http://www.arm.com/products/processors/technologies/trustzone/index.php

[19] A Tutorial on the Device Tree (Zynq) – Part I — xillybus.com. [online], [cit. 26.5.2015].
URL http://www.xillybus.com/tutorials/device-tree-zynq-1

[20] WebHome ¡ U-boot ¡ DENX. [online], [cit. 24.5.2015].
URL www.denx.de/wiki/U-Boot/

[21] Zynq-7000 All Programmable SoC. [online], [cit. 18.5.2015].
URL http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[22] Zynq UltraScale+ MPSoC. [online], [cit. 24.5.2015].
URL http://www.xilinx.com/products/silicon-devices/soc/
zynq-ultrascale-mpsoc.html

[23] Altera: *Cyclone V Device Handbook, Volume 3: Hard Processor System Technical Reference Manual*. 2014, [online], [cit. 19.12.2014].
URL http://www.altera.com/literature/hb/cyclone-v/cv_5v4.pdf

[24] ARM: *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. Armv7-a and armv7-r edition vydání, 2014, [online], [cit. 19.5.2015].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.
ddi0406c/index.html

[25] Corbet, J.; Rubini, A.; Kroah-Hartman, G.: *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005, ISBN 0596005903, [online], [cit. 26.5.2015].
URL http://lwn.net/Kernel/LDD3/

[26] Russell, R.: Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, ročník 42, č. 5, Červenec 2008: s. 95–103, ISSN 0163-5980, doi:10.1145/1400097.1400108, [online], [cit. 22.5.2015].
URL http://doi.acm.org/10.1145/1400097.1400108

[27] Texas Instruments: *OMAP3530 and OMAP3525 Applications Processors (Rev. H)*. 2013, [online], [cit. 25.5.2015].
URL http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=omap3530&fileType=pdf

[28] Xilinx: *Zynq-7000 All Programmable SoC Technical Reference Manual, UG585*. 2014, [online], [cit. 16.1.2015].
URL http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

Appendix A

CD contents

- `src/*`: Source codes and patches for the linux-socfpga and the OpenAMP.
- `doc/*`: Documentation source codes and final pdf file.

Appendix B

Captured virtio communication

```
rpmsg_virtio RX:
00 00 00 00 35 00 00 00 ....5...
00 00 00 00 28 00 00 00 ....(...
72 70 6d 73 67 2d 6f 70 rpmsg-op
65 6e 61 6d 70 2d 64 65 enamp-de
6d 6f 2d 63 68 61 6e 6e mo-chann
65 6c 00 00 00 00 00 00 el......
01 00 00 00 00 00 00 00 ........

struct rpmsg_hdr hdr = {
    .dst   = 0x35,  //53
    .len   = 0x28,  //40 - sizeof(struct rpmsg_ns_msg)
    .flags = 0x50,
    .data  = [...], //NS announcement message
};

NS announcement:
72 70 6d 73 67 2d 6f 70 rpmsg-op
65 6e 61 6d 70 2d 64 65 enamp-de
6d 6f 2d 63 68 61 6e 6e mo-chann
65 6c 00 00 00 00 00 00 el......
01 00 00 00 00 00 00 00 ........

struct rpmsg_ns_msg ns_msg = {
    .name  = "rpmsg-openamp-demo-channel",
    .addr  = 1,
    .flags = 0,
};
```

Listing B.1: Captured NS announcement message from remote firmware.

Appendix C

Matmul application output

```
 Demo Start - Demo rpmsg driver got probed
since the rpmsg device associated with driver was found !
 Create endpoint and register rx callback

 Master : Linux : Generating random matrices
 Master : Linux : Input matrix 0
 1  8  9  0  1  0
 8  3  7  9  0  1
 4  5  9  2  0  1
 7  9  8  6  4  5
 9  9  2  5  6  5
 1  9  3  1  5  4
 Master : Linux : Input matrix 1
 0  8  3  4  0  7
 4  8  6  5  6  4
 4  0  4  3  1  7
 5  9  4  7  4  2
 6  0  5  7  0  4
 6  9  6  6  2  5

 Master : Linux : Sent 296 bytes of data over rpmsg channel to
     remote

 Master : Linux : Received 148 bytes of data
 over rpmsg channel from remote
 Master : Linux : Printing results
  74   72   92   78   57  106
  91  178  112  137   63  140
  72   99   92   88   49  120
 152  227  181  197   96  194
 135  234  169  194   86  172
 107  125  122  124   69  106
```

Listing C.1: Matmul demo application Linux output.

Appendix D

Modified linux-socfpga source codes

```
IPI patch:
arch/arm/include/asm/hardirq.h
arch/arm/kernel/smp.c
include/linux/smp.h
kernel/irq/manage.c
kernel/smp.c

AMP patch:
arch/arm/mach-socfpga/platsmp.c
arch/arm/mach-socfpga/platsmp.h
kernel/irq/manage.c

GIC patch:
include/linux/irqchip/arm-gic.h
drivers/irqchip/irq-gic.c

remoteproc modifications:
drivers/remoteproc/soscfpga_remoteproc.c : added
drivers/remoteproc/remoteproc_core.c     : ioremap_nocache
```

Listing D.1: Modified Linux source codes.