Politecnico di Torino

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS
Master's Degree in Electronic Micro and Nanosystems

Master Thesis

DMA Support for the Sancus Architecture

Thesis advisor:
Prof. Guido Masera

Candidate:
Sergio Seminara

Research supervisors:
Dr. Jan Tobias Mühlberg
Ir. Jo Van Bulck
Ir. Pieter Maene

Academic Year 2018/2019

Abstract

Computing devices have a predominant role in our lives, changing most of our daily activities. With the advent of the IoT, more and more embedded devices are expected to be connected, with expected numbers around 20 billion of units for the 2020. Nevertheless, the increase in connectivity does not imply an enhancement of security. Programmable devices, especially connected ones, are at risk of being tampered with: recent history has shown many examples of malicious software attacks compromising their security. Considering that embedded device are required to be cheap in terms of resources they are implemented on low-end microcontrollers, which lack of any memory protection technique, making them unsuitable to implement solutions from the high-end world.

Hence, researchers have been recently focusing on strategies to provide them with security guarantees that hold even in case of an attacker with full control on the system, including the Operating System (OS). A promising solution is found in *Protected (software) Module Architectures* (PMAs): security architectures that can execute protected code in an isolated area of the memory, inaccessible to other software. PMAs can support secure execution of small portion of code, the software modules (SMs), even on devices that are, e.g., malware infected.

The target architecture of this thesis is Sancus, an open-source, hardware-only PMA, designed for lightweight embedded devices. The objective of the work is to extend the architecture with Direct Memory Access (DMA) support, a feature that provides peripherals with a secondary channel to access the memory without involving the CPU, and to explore what does it entail from a security prospective. Usually lightweight PMAs do not support DMA, as the benefits coming from its inclusion on the system do not comply with security properties of these architectures: protected memory isolation and confidentiality are no longer guaranteed if a secondary channel, that directly access the memory independently from the CPU, is provided. Main achievements of the thesis are to extend the Sancus 2.0 architecture with DMA support, showing how this affects its security properties and providing a secure way to implement DMA. From the discussion of the upcoming chapters, two solutions stood out: the first one is currently implemented on Sancus, and it consists in entirely excluding DMA from accessing protected memory, preserving SMs security guarantees. The second theoretical solution aims to provide the software modules with DMA functionalities, by allowing some security guarantees for a confined regions inside protected memory. The latter solution is discussed in details and some design options are provided, whereas its implementation is deferred as future work.

Sammanfattning

Datorer har en dominerande roll i våra liv och förändrar de flesta av våra dagliga aktiviteter. Med tillkomsten av IoT förväntas fler och fler inbyggda enheter anslutas, med förväntat antal cirka 20 miljarder år 2020. Ändå innebär ökningen av anslutning inte en ökning av säkerheten. Programmerbara enheter, särskilt anslutna, riskerar att bli manipulerade: de senaste erfarenheterna visar många exempel på skadliga programattacker som äventyrar säkerheten. Inbäddade enheter måste vara billiga när det gäller resurser. Därför implementeras de på low-end mikrokontrollers, som helt saknar stöd för minnesskydd vilket gör dem olämpliga för lösningar från high-end-världen. Därför har forskare nyligen fokuserat på strategier för att ge dem säkerhetsgarantier som håller även i händelse av en angripare med full kontroll på systemet, inklusive operativsystemet (OS). En lovande lösning finns i *Protected (Software) Module Architectures (PMA)*: säkerhetsarkitekturer som kan exekvera skyddad kod i ett isolerat område i minnet, otillgängligt för annan programvara. PMA kan stödja säker exekvering av liten del av koden, mjukvarumoduler (SM), även på enheter som är infekterade och innehåller skadlig programvara.

Målarkitekturen för den här avhandlingen är Sancus, en öppen källkod endast för maskinvara-PMA, avsedd för inbyggda lättviktsenheter. Syftet med arbetet är att utöka arkitekturen med DMA-stöd (*Direct Memory Access*), en funktion som ger perifera enheter en sekundär kanal för att komma åt minnet utan att involvera CPU, och att undersöka vad det medför från ett säkerhetsperspektiv. Vanligtvis ger inte lättvikts-PMA stöd för DMA, eftersom fördelarna som följer av att den ingår i systemet inte överensstämmer med säkerhetsegenskaperna hos dessa arkitekturer: skyddad minnesisolering och konfidentialitet garanteras inte längre om en sekundär kanal, som har direkt åtkomst till minnet oberoende av CPU, tillhandahålls. Huvud-resultatet i avhandlingen är att utöka Sancus 2.0-arkitekturen med DMA-stöd, vilket visar hur det påverkar dess säkerhetsegenskaper och ger ett säkert sätt att implementera DMA. Två lösningar presenteras: den första är för närvarande implementerad på Sancus och består helt och hållet av att DMA inte ska få tillgång till skyddat minne och bevarar säkerhetsgarantier från SM. Den andra teoretiska lösningen syftar till att tillhandahålla mjukvarumoduler med DMA-funktioner, genom att tillåta vissa säkerhetsgarantier för en begränsad region inom det skyddade minnet. Den senare lösningen diskuteras i detalj och vissa designalternativ ges, medan dess genomförande skjuts på framtiden.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU	Arithmetic Logic Unit
ASM	Algorithmic State Machine
CPU	Central Processing Unit
CU	Control Unit
DE	Data section End
DMA	Direct Memory Access
DMA_PE	DMA Protected End address
DMA_PS	DMA Protected Start address
DMAC	Direct Memory Access Controller
DS	Data section Start
ELF	Executable and Linkable Format
FIFO	First In First Out
FSM	Finite State Machine
IoT	Internet of Things
IP	Infrastructure Provider
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MAB	Memory Access Bus
MAC	Memory Authentication Code
MAL	Memory Authentication Logic
MMIO	Memory-Mapped Input/Output
MMU	Memory Management Unit
PMA	Protected Module Architecture
PSA	Protected Storage Area
SM	Software Module
SP	Software Provider
TCB	Trusted Computing Base
TE	Text section End
TS	Text section Start

Chapter 1

Introduction

1.1 History

Nowadays, computing devices have a predominant role in our lives, unveiling us new services that irreversibly changed most of our daily activities. All of this is possible since we rely on the providers of those services to securely process our data on our devices. However programmable devices are at risk of being tampered with, especially connected ones: recent history has shown that these devices are subject to malware infections and, in general, software attacks that compromise integrity and data confidentiality: in one word, the devices security [15]. Relevant experience has been gained for securing established high-end systems, such as desktops and servers, which have been on the market for a long time. In order to protect them from malicious exploits, some solutions arose, trying to fix the flaws as they are exploited by attackers.
With the advent of IoT future isn't brighter. Promoted by the enhancement of the technology, more and more embedded devices are expected to be connected, wit expected numbers around 20 billion of units for the 2020 [2, 3, 16]. Nevertheless, the increase in connectivity and the impact of devices on our society do not entail an enhncement of security. Critical software vulnerabilities have been disconed in home appliances [31], cars [38] and even industrial sites [47], each implying different level of threats: an hacked fridge could leak some personal data, at worst, but when it comes to cars, industries or medical devices, the threat becomes far more worrying [30].

Therefore, on the past years, researchers have tried to answer the open question of how to secure networked programmable devices, and to allow a stakeholder to assess the trustworthiness of a computing device. Important solutions are virtualization of the memory in combination with processor privilege levels, and the im-

plementation of a memory-safe virtual machine. Both relying on a secured software layer, either the operating system (OS) or the virtual machine implementation, the two solutions provide isolation of the software running on the processor. The first technique takes advantages of hardware support, generally a Memory Management Unit (MMU), to provide an abstraction of the storage resource so that the main memory appears to be larger than it really is. The operating system, which also relies on protection rings, assigns separate memory space to running processes and guards the interaction among them. The second technique involves the uses of a memory-safe virtual machine, where software is deployed in memory-safe byte-codes and a security architecture, usually an hypervisor, guards the interaction of different software modules [45]. However, with classical solutions it is non-trivial to provide an attestation mechanism, which a remote stakeholder can rely on to check that a specific software module is running untampered with on a remote device. Finally, while software-based solutions have the advantage of compatibility, they also negatively impact on performances and, most important, they cannot protect from system-level attacks: once the relying software layer - OS or hypervisor - has been compromised [49], security of the system can no longer be guaranteed: a malicious operating system can, in fact, allow manipulation of the software, breaking the root of trust. To ensure resilience even in these cases, hardware-based solutions have to be used.

Embedded device are required to be cheap in terms of resources such as chip area, chip complexity, power consumption and performance, thus they are implemented on low-end, resource-constrained microcontrollers. This makes them unsuitable to implement established solutions from high-end devices world, which require a more complex, thus expensive, architecture to be integrated.
A promising solution is found in *Protected (software) Module Architectures* (PMAs): security architectures running independently from a classic operating system, that can execute code in an isolated area of the memory. The isolation is not built on the operating system, thus improving the security [45]. The idea of PMAs originally arose from the need to split complex software into smaller *protected modules*, isolated between them, whose correctness would have been easier to verify [34]. The use of PMAs also provides a secure way to support *remote attestation* where classical solutions could not. Remote attestation is the capability of a stakeholder to remotely check that a specific software module is running untampered with on a remote device. In general, a device is said to be *trusted* if it behaves as expected even when tampered with at the software level [33], i.e. even when an attacker gains control on the operating system [30].
Currently different PMAs are available, each coming with a different cost in terms of hardware onhead and performance. Some provide isolation of application on high-end devices, for example the Intel SGX [13] which extends x86 architectures

and whose cost is not considered lightweight at all. Others leverage Memory Encryption Engines to encrypt/decrypt sensitive code and data when leaving/entering the main memory, like for example Atlas [29], or make use of lightweigth Memory Protection Units to enforce an execution-aware memory access control, as done in TrustLite [26]. Finally, architecture like SMART [14] aim to provide only some security guarantees with the minimal set of hardware changes. Program-counter based memory is a memory protection technique whose aim is to provide isolation between software modules running on the same node, but that do not necessarily trust each other [45]. PC based memory access proposes memory access control based on the current value of the program counter, resulting in being a low-cost alternative to the virtualization of the memory or the use of a memory-safe virtual machine.

1.2 Problem Statement and Goal of the Thesis

The combination of PMAs and hardware implementations of the program-counter based memory access control is one way to extend protected module idea to the low-end microcontroller world, achieving hardware-based trusted computing. An architecture of this kind enhances security guarantees of the system by having the root of trust buried in the hardware, which is considered to be immutable from attackers [30]. For this reasons it is preferable to develop on hardware-based architectures rather than software-based, although some software-based architectures with interesting results have been proposed [4, 24, 32, 43, 46].

The target architecture of this thesis is Sancus[1], an open-source,[2] hardware-only PMA designed for lightweight embedded devices. By leveraging an hardware implementation of the PC based memory access control, Sancus guarantees the secure execution of protected applications. The root of trust is, thus, buried into hardware, enhancing the security guarantees of the architecture. It allows external peripherals connection but it natively does not support Direct Memory Access.

Direct Memory Access (DMA) is a feature of CPUs that allows hardware subsystems to directly access the memory, without the participation of the Control Unit (CU). First proofs of DMA integration on computing systems date back to

[1]Designed in 2013 by Noorman et al., Sancus is an ongoing project of imec-DistriNet and COSIC research groups of KU Leuven, that reached its second version.
Sancus 2.0 guarantees a set of security properties, such as software isolation as well as with local and remote attestation. Moreon, it supports secure linking and secure communication . For more details you can visit *https://distrinet.cs.kuleuven.be/software/sancus/index.php*

[2]More details about Sancus source code are provided in section A.1

the Intel 8085 (1976) Intel 8 bit microprocessor, or the IBM Personal Computer (1981), both provided with the DMA Controller (DMAC) chip known as Intel 8237 [22]. By means of DMA, devices can transparently access the main memory unburdening the CPU from I/O loads. This latter is then free to handle other operations, while the data transfer is happening. In this sense, DMA speeds up the system.

Generally PMAs do not support DMA. The reason is that an attacker with DMA capabilities can tamper with any location of the system memory at run-time, as DMA bypasses any MMU-like control. Program-counter based memory access control is, in fact, enforced on the memory access bus (MAB) alone, i.e. on every access to the memory going through the CPU. What if the untrusted element resides outside CPU domain? What if there was a way to directly access the memory, bypassing any CPU control, so that no violation is raised on illegal accesses? Hence, a protection mechanism has to be provided if DMA is going to be implemented on a PMA, in order to prevent disastrous outcomes.
Some attacks have successfully shown that it is possbile to exploit DMA capabilities of ad-hoc compromised devices [44, 11, 41, 12, 37, 6] to tamper with the host system, as reported in [42].

The goal of this thesis is, first, to extend Sancus 2.0 architecture with DMA capabilities, then, to show that such a functionality breaches security guarantees of the system. Lastly some ideas to secure Sancus SMs from DMA attacks have been explored; the aim it to extend the same memory hierarchy that rules the internal memory accesses to DMA peripherals, and to keep the hardware onhead as small as possible. Section 3.2, *"Exploitation of Naive DMA Support"*, guides the reader through a simple attack targetting the Sancus architecture. It leverages the disruptive capabilities of a naive DMA implementation, in which no memory access control is enforced on the DMA bus, to leak secret data from a Sancus software module and to inject malicious content in the data section. section 6.7 reports the outcomes of a similar attack when carried on two different Sancus implementations: one equipped with a direct implementation of the DMA interface, and the other supporting secure DMA. The provided results underline the importance of providing a secure DMA channel on Protected Modules Architectures.
Hence, the main contributions of the thesis can be summarised in:

1. Show that the inclusion of DMA support on Sancus 2.0 breaks isolation and confidentiality guarantees for the architecture (section 4.4), and underline the need to validate DMA accesses independently from the PC (subsection 4.5.2).

2. Implement a simple solution to preserve Sancus security guarantees, by preventing DMA to access any protected memory location (subsection 4.5.3).

3. Propose a further conceptual solution to extend software modules capabilities with DMA functionalities, at the expense of relaxing isolation and confidentiality guarantees for a confined region inside modules protected memory (subsection 4.5.4).

Chapter 2

Background

Computing devices play a predominant role in our lives and daily activities. Basing on the enormous number of data available, they can provide us with an entire new level of personalised services. With the advent of Internet of Things (IoT) the phenomenon is only going to increase, with embedded devices becoming omnipresent and interconnected, handling safety-critical and privacy-sensitive data. Yet, the increase in connectivity does not entail an enhancement of the security these devices guarantees. To minimize production costs and power consumption they are usually implemented on lightweight architectures, which lack of hardware support for classical solutions from high-end devices world, such as the virtualization of the memory, in conjunction with processor privilege levels, or the use of a virtual machine to run critical code in a safe environment.

The increasing trend of connecting computing devices opened to malware infections and, in general, software attacks that compromise systems security [15]. Computer security is a growing field of the broader information security, that aims to protect computing systems and the data that they store or access to. During last decades, a rich body of experience has been acquired in securing high-end computing devices, such as desktops and servers, through the classical solutions aforementioned. Unfortunately, these solutions do not comply with the resource-constrained, low-end devices world, characterised by a very simple memory hierarchy in which the use of Memory Management Units (MMUs) is not contemplated.

A recent line of research focused on Protected Module Architectures (PMAs) as alternative to tackle the problem of security by means of a fine-grained memory access control. PMAs can give user security guarantees about the behaviour of software running on them, achieving the main goal of *trusted computing*, which is to protect applications and users from malicious exploitation. When a system keeps behaving as expected even when tampered with, i.e. even when an attacker gains control on it, it is said to be *trusted* [30]. An important part of trusted

computing is to provide security guarantees even in case the OS is tampered with. Unfortunately, software-based solutions are inadequate in achieving this goal, as an attacker can always manipulate software if the OS is not trusted. In order to provide strong guarantees even in this eventuality, hardware-only based architectures have to be considered. For these systems the root of trust is said to be buried in the hardware and is more resilient to malicious attacks than software solutions, to the extent that hardware is considered to be immutable.
Finally, the minimization of the Trusting Computing Base (TCB) has been a paramount goal of computer security, since the start of the field, and many hardware-only PMAs manage to provide security guarantees with a small TCB.

The TCB is defined as *"the set of the hardware and software components which are considered critical for the security of the whole system. The TCB is designed so that, even in case of tampering with other parts of the system, the device cannot misbehave"* [30].
A more detailed explanation of the PMAs together with the protection techniques used to provide modules isolation is outlined in the following section 2.1.

2.1 Protected Module Architectures

Protected architectures run independently of the Operating System (OS) and allow the secure and isolated execution of software modules [30]. The idea of PMAs originally addressed the need to split complex software into smaller protected modules, isolated between them, whose correctness would have been easier to verify [34]. Several prototypes have been developed both for embedded systems – such as Sancus [39], TrustLite [26], TyTAN [8] or Soteria [21] – as well as for high-end systems – such as Intel Software Guards eXtensions (SGX) [35] or Iso-X [17] – ensuring modules isolation.

A countermeasure to software complexity is to use a divide-and-conquer approach, where big bunches of code are divided in smaller, and more maintainable, software modules (SMs). A SM essentially consists of two separated text and data sections. The former contains the module protected code and can be entered only via few predefined addresses of memory locations, called entry points. Each software module should maintain its own private call stack, for example by saving it in its private data section. The latter contains private data of the module and can be accessed (read or written) only when the program counter is pointing within the code section of the corresponding module. The access rules are enforced by the PMA on which the SM is running.
By combining the PMAs idea with the program-counter based memory access control it is possible to extend memory protection mechanism to low-end embedded devices.

from \ to	Protected			Unprotected
	Entry point	Code	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

Table 2.1: Access control rules to the memory in PC based memory access control. Access rights are shown in a Unix notation, indicating how code executing in the *'from'* may access the *'to'* section [39].

2.2 Program-Counter Based Memory Access Control

Program-counter based memory access control is a memory protection technique based on the current value of the program counter: depending on where the code is being executed, different access rights to the memory are enforced. Code executing from unprotected memory regions has no access rights to protected memory, except for execution rights on the entry point of a software module. Accesses from protected to unprotected memory are granted with full access permissions. Notice that only code executing from within a software module can access to its text and data sections. In this way, module isolation is guaranteed, as only SMs have direct access to their own text and data sections. When more than one software module are loaded on the same PMA, the rules for accessing its private sections treat all other modules as unprotected memory regions [45]. Table 2.1 shows the access rights enforced for code executing in protected and unprotected memory locations.

Notice that PC based memory access control is sufficiently strong to preserve all the isolation guarantees that modern programming languages can provide, such as information hiding and encapsulation - which is basically what is ensured when declaring a private variable in high level programming languages, as C++ or Java, for example. However, all these guarantees are typically lost when the program is compiled, as an attacker operating at machine code level can always break isolation and confidentiality of the compiled code by directly tampering with it in the system memory. Interestingly, it has been shown [1, 40] that source code securely compiled to a PMA maintains the source code level abstractions even against machine code level attacks. For hardware-only PMAs these guarantees even hold in case of a compromised OS.

2.3 Sancus 2.0

2.3.1 Overview

Program-counter based memory access can be implemented in different ways; the target architecture of this thesis is Sancus, a hardware-level implementation that provides strong isolation, remote attestation, as well as a series of security properties as secure communication, secure linking, confidential software deployment and hardware breach confinement.

System model The architecture addresses the problem of securing an infrastructure made up of a set of interconnected low-end microprocessor-based systems, which are referred as *nodes* N_i. The infrastructure provider *IP*, owner of the network, allows some *software providers*, with public IDs SP_i, the access to the nodes, giving them the possibility to deploy *software module SM* onto them. The TCB on the networked devices is hardware-only, specifically each node does not trust any software, including the operating system running on it.

Attacker model The attacker is consider to have two critical capabilities. First it can manipulate all the software on the node, including tamper with the OS, and it can act as a software provider by deploying malicious software modules on the nodes. Second, it can control the communication network used by the software provider and the nodes to communicate with each other; attacker can sniff the network, modify traffic and mount man-in-the-middle attacks.
However, it cannot break cryptographic primitives and does not have physical access to the hardware: it cannot modify it, nor connect probes, nor disconnect components, and so forth.

Cryptographic primitives and node keys The architecture design relies on three cryptographic primitives:

- a classical cryptographic hash function to compute digest of data [36];
- a key derivation function [36], used to derive a key from a master key ans some diversification data:

$$K_{Master,Data} = kdf(K_{Master}, Data)$$

- a pair of *authenticated encryption* and *authenticated decryption* with *associated data*. The encryption function takes as input a key K, a plaintext P and associated data D, and produces in output a ciphertext C of the plaintext,

Key	Creation	Accessible by	Saved[1]
K_N	Random	IP, N	$\surd$
$K_{N,SP}$	$kdf(K_N, SP)$	SP	
$K_{N,SP,SM}$	$kdf(K_{N,SP}, SM)$	SM (indirectly)	$\surd$

Table 2.2: Overview of the keys used in Sancus, how they are created, stored, and who can access them [39].

plus a Message Authentication Code (MAC) T on both the plaintext and the associated data [36].

$$\begin{aligned} C, T &= \texttt{aead-encrypt}(K, P, A) \\ P &= \texttt{aead-decrypt}(K, C, A, T) \end{aligned} \tag{2.1}$$

The cryptographic keys are handled by the many parties of the network: the IP shares a symmetric key with each of its nodes, the *node master key* K_N. This is saved in an internal register of the CPU, accessible only indirectly through specific processor instructions. The software providers allowed to deploy modules on a specific node are provided with the intended software provider key $K_{N,SP}$, computed by the IP with through the key derivation function $kdf(K_N, SP)$. Notice that Sancus nodes host an hardware implementation of the *kdf*, hence they can compute $K_{N,SP}$ by themself. The key derivation function used in the current Sancus implementation is the *SpongeWrap*. Finally, nodes can compute the symmetric key $K_{N,SP,SM}$ relative to the software module SM. The key is stored in a protected area of the processor, inaccessible from the software. Consider that software providers can compute keys relative to their own modules, as it received the $K_{N,SP}$ from the IP, and it knows the identities of the modules it is loading on N.

Software modules *Software modules,* SM_i are binary files composed of a *text section* and a *data section*. The former contains code and constants of the software module and can only be accessed by jumping to one of its fixed entry points; the content of text section is enforced to be read only, after memory protection is enabled. The latter contains runtime data of a module, such as the call stack which is included in the data section to avoid leaks. The Sancus architecture comes with a specific compiler that automatically handles the inclusion of every runtime metadata in the data section.
When a software module *SM* is loaded in the main memory, the processor saves the *layout* of the module in a protected storage area inside the CPU: the Protected

[1]Saved in the Protected Storage Area of the node CPU.

Storage Area (PSA). The layout consists of the start and end addresses of both text and data sections. The *identity* of the module is obtained, by computing the hash of the layout and the content of the text section of a module, and it's stored in the PSA.
Finally, the symmetric module key $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$ is computed and stored in the protected area, to be used with cryptographic functions as assurance of module integrity. In this regard, notice that the software provider *SP* can also compute $K_{N,SP,SM}$, since it received $K_{N,SP}$ from *IP* and knows the identity of the deployed module *SM*. Therefore $K_{N,SP,SM}$ is to encrypt data before sending it on the untrusted communication network, and decrypt them once received, securing the communication. Data decryption only works if both the players computes the same key $K_{N,SP,SM} = kdf(K_{N,SP}, SM)$: in other words, the mechanism works as long as the software module *SM* is not compromised before the key is computed – i.e. before the protection is enabled. After that moment modules isolation assures that the content of modules private sections is inaccessible from the outside the modules (here it is implicitly assumed that software provider has not been tampered with, as well as $K_{N,SP}$ has been correctly computed by *IP* and received by *SP*).

2.3.2 Security Properties

Here follows a summary of the security guarantees that the Sancus architecture achieves, as presented in [39]

Software module isolation The isolation property can be considered the fundamental property of the system: software modules run isolated on a node if their protected section cannot be accessed from outside the module themself. In Sancus this is achieved through a program-counter based memory access control [46], which implies that control flow enters into a module text section only by calling the entry point of that specific module. Moreover, the module data section is only accessed from code executing in the text section of the intended module. Notice that the protection is enabled only after the execution of the `protect` instruction:

```
protect(layout, SP)
```

until that moment, the content of a SM can be modified, by an attacker with full software capabilities. Any modification to the text section, occurring before the `protect` instruction is called, alters the module identity. This causes a mismatch between the key computed and stored into the node $K'_{N,SP,SM}$ and the the key computed by the software provider $K_{N,SP,SM}$ (Table 2.2), with consequent failure of successive attestations. On the contrary, modifications to the data section

are not a concern since the `protect` instruction zero initialize the content of the section, when called. The processor `protect` instruction does:

- check that the layout of the new module does not overlap with any existing modules and, if this is the case, continue with the deployment by storing its layout in the PSA of the processor;
- enable the memory access control on the loaded module;
- compute the module key $K_{N,SP,SM}$ basing of the layout informations, and store it in the PSA of the node;

The only way to lift the memory protection is through the processor instruction `unprotect(continuation)` from the inside of the module to be deactivated. To prevent any leakage, the instruction also clears module's code and data sections. Since the `unprotect` instruction is itself part of module's text section, a pointer to the code where the execution is to be resumed must be provided as *continuation* argument [39].

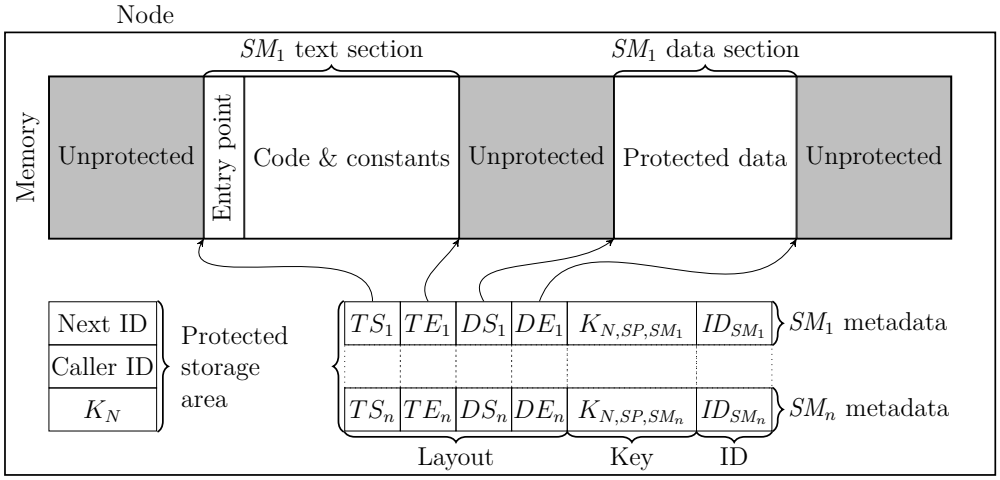

Figure 2.1: Representation of a Sancus node with a software module SM_1 loaded. The content of the processor Protected Storage Area (PSA) is shown: it's important to underline that node key K_N, together with all SMs keys K_{N,SP,SM_i}, are stored in the PSA, which is not mapped into the system memory, and indirectly accessed by the software through specific processor instructions. Thus, they cannot be leaked. This is a peculiarity of the Sancus architecture in contrast to other solutions which leak the key, as it happens, for example, with Intel SGX in the Foreshadow attack [9].

Remote attestation Remote attestation is a crucial property for a networked infrastructure as the one Sancus has been thought for. Being able to remotely attest the identity of software modules executing on a remote node is a fundamental requirement for a software provider, since it guarantees that the intended module is correctly executing. In order to achieve it, two cryptographic primitives - `encrypt` and `decrypt` - are used (Equation 2.1). Due to the use of a freshness guarantee, they are resilient to replay attacks. The process of remotely attesting a *SM* running on a node *N* starts with the software provider *SP* sending a fresh nonce *No* to the node. The nonce is passed by the untrusted software running on *N* to the module, as argument of the encryption function to be called. Then, the module *SM* encrypts the received nonce N_0 using its module key $K_{N,SP,SM}$, as shown in Equation 2.2. By discarding the output cyphertext *C* a Message Authentication Code *T* is obtained, which solely refers to the data N_0.

$$C, T = \texttt{aead-encrypt}(K, \{\}, D) \tag{2.2}$$

The computed MAC is then sent to the *SP*, which will compute a copy by means of its own copy of the module key $K_{N,SP,SM}$. If the two tags coincide, it is a proof that the software module *SM* is running untampered on the node *N* at this point of time. Notice that this property deeply leverages on the assumption that, after protection is enabled, SMs protected sections cannot be modified.

Secure communication By relying on the same mechanism used for remote attestation, it is possible to set a secure communication between software modules and software providers. Whenever a software provider *SP* wants to receive some data from a module *SM*, running on a node *N*, it sends a nonce *No* and, possibly, input data *I* to the intended module. The request is received by the untrusted software on the node and passed to the software module, which executes the code and produces an output *O*. The *SM* encrypts the output through the encrypt function $C, T = aead - encrypt(K_{N,SM,SP}, O, No||I)$, and sends the resulting cyphertext *C* and the tag *T* to the *SP*, on the unsecure network. Since *SP* knows the key $K_{N,SM,SP}$ (remember the key management of Table 2.2) it can verify the tag and `decrypt` the output. In this way the software provider has strong assurance that the output *O* has been produced by *SM* on node *N*, given nonce *No* and input *I*.

Secure linking High assurance of calling the intended module must be provided when a software module tries to link with another module, on the same node. In order to do so, the module SM_1 that wants to connect to module SM_2, needs to include SM_2's identity in its text section[2]. In order to allow secure linking, the

[2]Note that if SM_2 wants to connect to SM_1 too, this method creates a circular dependency between their identities. A solution is that the software provider does not include the identity

processor is provided with two special instructions:

```
attest(address, expected hash)
```

The `attest` instruction proceeds by:

- verifying that a module, with protection enabled, is loaded at the provided *address*;
- computing the identity of that module, by using the cryptographic primitive hash function of page 10 on that module's text section and layout;
- comparing the result with the *expected hash* parameter of the instruction;
- returning module's ID in case the hashes were equal, zero otherwise;

Since the authentication process includes the computation of an hash, it is rather time consuming and expensive. Therefore the processor assigns sequential IDs to the loaded modules; the IDs are ensured to be used just once within one boot cycle[3].
Hence, the IDs of verified modules are used to speed up subsequent attestations: it suffices to check that the same module with the same ID is still loaded at that address. The processor function

```
get-id(address)
```

checks that a protected module is loaded at *address* and returns the ID of that module. For caller authentication the node keeps track of the previously executing module by recording its ID in a register *"Caller ID"* of the PSA. The register is updated every time the execution flow enters in a different module. Modules can attest the caller identity though two instructions `attest-caller` and `get-caller-id`, whose usage is analogous to, respectively, the `attest` and `get-id` instructions, except that they implicitly use the caller ID.

Confidential loading Module's content is indeed protected, but only after protect instruction has been enabled; up to that moment an attacker can easily read the text section, without being noticed[4]. A software provider can deploy encrypted

of one of the two module in the text section, but it securely sends it after deployment has been done, storing in the data section [39].

[3]This can be easily done by having the ID to be used for the next module stored in a register *"Next ID"* in the PSA. The value of the register is automatically incremented every time a new module is enabled, and it generates an error when it overflows.

[4]Remember from page 12 that if the attacker modifies the text section, it causes a change in the module's key, which is detected in subsequent attestation; data section is not worrying since its content is zero initialized after protection has been enabled.

modules that will be decrypted in place before loading. In this way the content of a module is inaccessible for an attacker. Therefore a second way of using protect instruction is provided:

```
protect(layout,SP,MAC)
```

The MAC is the authentication encryption tag provided by the *SP* which encrypted the text section of the module, with the key $K_{N,SP}$.
The instruction behaves exactly the same as before, except that it decrypts the module before calculating the module key. If the integrity check using the given MAC fails, the text section is cleared and the protection disabled.

Hardware breach confinement In the possibility that an attacker manages to breach the hardware protection of a node, the breach should be confined to the node in question and not affect the rest of the infrastructure. Since every node possesses its own private key, a compromised node does not provide to the attacker any further information regarding the *other* nodes. Therefore, an attacker could impersonate software modules form the compromised node, but it cannot extend the breach to other nodes.

Memory access violation When a memory access violation is detected the architecture simply resets the processor, clearing the memory to prevent any leak. This has the advantage of being a secure mechanism, as no information can leak, and simple at the same time. A big disadvantage is that it can have a bad impact on availability of the node: a bug or a malicious software may cause the node to keep resetting and clearing its memory.

2.3.3 Attacker Model

The attackers considered in Sancus are assumed with the following capabilities:

- Attackers can manipulate all the software on the node. Specifically they can act as a software providers and deploy malicious modules on the node, tamper with the operating system or even install a completely new one.
- Attackers can control the communication network that is used by the software providers and nodes to communicate between each other. Independently of the security of the communication channel, which is out of scope, they can sniff the network, modify traffic, or mount man-in-the-middle attacks.

- Attackers cannot break cryptographic primitives, but they can perform protocol-level attacks.[5]
- Attackers cannot have physical access to the hardware, thus they cannot probe the memory bus nor disconnect components, and so forth.

2.3.4 Implementation

2.3.4.1 Hardware Implementation

Sancus nodes are based on the open source implementation of the TI MSP430 corefrom the Open Core project [18]. In order to provide nodes with secure functionalities some modifications are carried out: first of all, modules layout informations as well as the modules keys $K_{N,SP,SM}$ and identities are saved in internal CPU registers, thus only a finite set of modules can be deployed. Second, several instantiations of the *Memory Access Logic (MAL)* circuit are created, one for each module that can be instantiated in the system N_{SM}. Each circuitry implements the program-counter based memory access control. It has five inputs: current and previous program counters (PC), current address of the memory access bus (MAB) as well as memory access bus enable and write flags. It compares the PC and the MAB with the start and end addresses of text and data sections of a SM, when the protection *EN* bit is asserted. In case of illegal access to the memory a violation signal *sm_violation* is raised. The MAL circuit is instantiated N_{SM} times, with N_{SM} being the maximum number of software modules that can be deployed on the node. You can find a representation of MAL circuit in Figure 2.3.

OpenMSP430 version in use in Sancus it is not natively provided with a DMA interface. The first tasks of this thesis is to upgrade Sancus by extending it with DMA interface functionalities as provided in the last version of openMSP430[6]. In order to do so, substantial modifications to the memory backbone carried out, as the DMA interface signals have to correctly interact with memory accesses of fetch, decode and execute CPU phases. Further details can be found in chapter 5.

2.3.4.2 The Compiler

Although software developers can rely on Sancus security functions to create protected modules. However doing this correctly still requires doing this correctly still requires carefulness to not introduce software bugs that may invalidate the system security. Firstly a module must have only one entry point. Secondly each

[5]Protocol-level attack is defined as the exploitation of specific feature or implementation bug of some protocol installed at the victim for consuming huge amount of its resources.

[6]At the time of writing, the openMSP430 core is realeased with version r211.

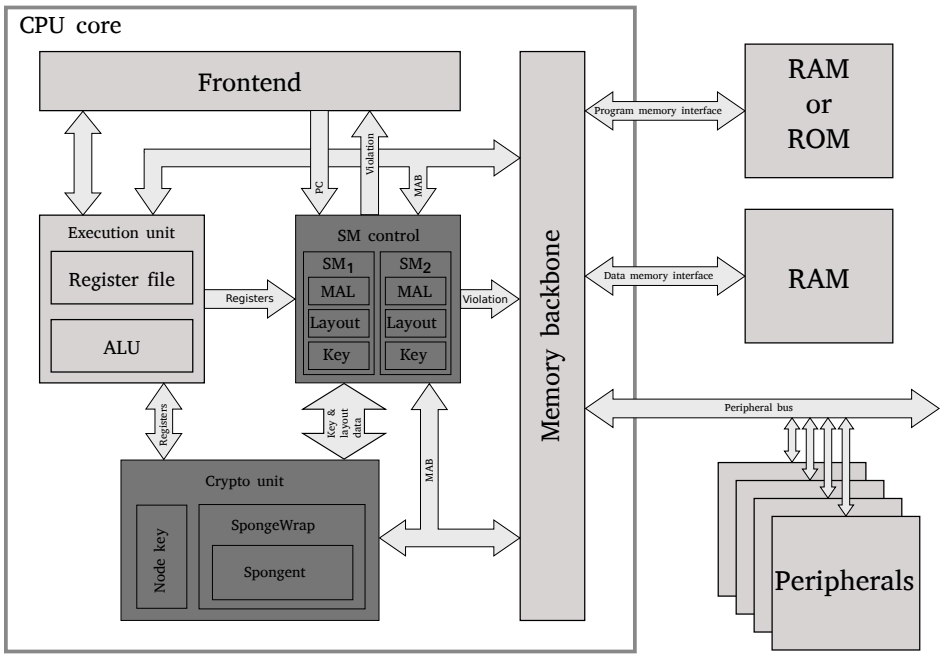

Figure 2.2: Overview of the hardware block constituting a Sancus core. Lightly shaded blocks are part of the original openMSP core, whereas darker shaded ones are specifically added for Sancus. Remember that the number of SMs N_{SM} is set when synthesizing the core. Notice how the SM control unit takes the program counter (PC) and the memory access bus (MAB) as inputs for memory access logic (MAL) circuits [39].

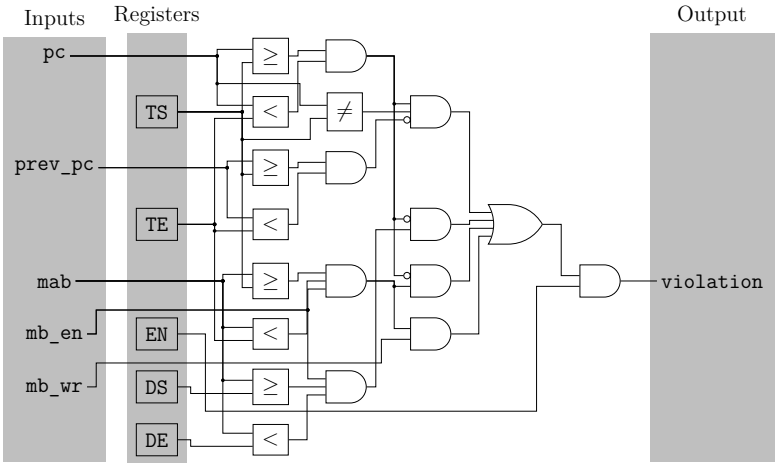

Figure 2.3: Schematic of the Memory Access Logic (MAL), the hardware used to enforce the memory access rules for each protected module [39].

module need to implement its own call stack to avoid leakage of sensitive data to unprotected code. Thirdly, exiting from a module, or calls to another one, requires specific instructions to be carried out in the right order. The compiler assures that everything is correctly handled, with no further burden on the programmer side. To deals with these low level details, Sancus authors implemented a compiler extension based on LLVM [19], together with support library that offers API to perform some commonly used function like calculate a MAC of data. "The compiler compiles C standard files, and it uses Clang [19] as compiler frontend. To benefit from Sancus, developers only need to indicate which functions should be part of protected module, which should be the entry points and what data should be in the protected section. This is done by means of compiler attributes `-SM_FUNC`, `SM_ENTRY` and `SM_DATA-` respectively used to annotate functions, entry points and modules private data." [39].

Entry points The compiler implements multiple logic entry points on top of the single physical entry point of a software module, by means of a jump table. The compiler assigns a unique ID to every logical entry point; when code tries to jump to one of these entries, its ID is stored in a register and the jump is redirected to the correct physical entry point of the module. The code, then, jumps to the correct called function based on the ID passed in the register.

Stack handling The compiler also takes care of reserving a space in the protected section of the module for the stack. The first time a module is entered the stack pointer is loaded with the address of this start location of the module stack. Stack pointer of the module is saved on exits, for further usage.

Exiting Modules and Secure Linking When exiting from a module any register that is not holding a parameter to be returned is cleared, to avoid leakages. Calls to protected modules automatically verify the caller ID: in order to do so a software provider needs provide its key to the compiler.

2.3.4.3 Software Stack for Deployment

Deploying a module is a sensitive operation: module's identity, thus its key, depends on module's load address on the node. These *SP* must be aware of those addresses in order to correctly compute $K_{N,SP,SM}$. Enforcing static loading is not a scalable solution given that the target system supports module loading from different software providers. Hence the need to implement a software stack to dynamically deploy modules on a node. The process starts with the *SP* creating an Executable and Linkable Format (ELF) file of the *SM* and sending it to the

node *N*. The host software on the node receives the files, finds a free memory area to load the *SM* and relocates it using a dynamic ELF loader. Then hardware protection is enabled and a symbol table is sent back to the *SP*. The symbol table contains the addresses of any global functions along with the load addresses of all protected modules on the node *N*. By using the symbol table the software provider can reconstruct the exact same image of *SM* as loaded on the *N*, which can be use for the computation of the module key.
Loading of encrypted modules requires an extra step, since the dynamic loader need to inspect and update parts of the text section, which is inaccessible in case of encrypted modules. First the *SP* sends a request to the *N* specifying the size of the encrypted module it wants to load. Then the host software allocates memory for the module's sections and answers back with an handle identifying the memory allocated location, and the symbol table. *SP* links locally the *SM* and sends the resulting image back to *N*, together with the memory handle. The host software on the node loads it in the pre-allocated memory sections and enables the protection. After the *SM* has been deployed, the software on the node provides an interface to be able to call its entry point. From this moment the module is fully operative on the node, thus the *SP* can use the provided interface to start operations on the module, such attesting its status or performing I/O accesses to its data. The interface is used to provide the *SM* with the identities of the other modules, it can securely link to. To this end, the *SP* computes the identity hashes from the symbol table, which contains the image of the loaded modules or, if the module to be loaded belongs to a different software provider, it receives them from the respective providers. Finally the hashes are encrypted with the $K_{N,SP,SM}$ and sent to the node.

2.4 Direct Memory Access (DMA)

2.4.1 Overview

Direct Memory Access (DMA) is a feature of CPUs that allows hardware subsystems to directly access the memory, without the participation of the Control Unit (CU). Without DMA mechanism, CPU would be fully occupied during I/O operations, thus unavailable to perform other tasks. In this sense, DMA speeds up the system, unburdening the CPU from I/O loads. In general DMA traffic is supervised by a DMA Controller (DMAC): the device that wants to start a DMA operation has to configure the DMAC internal registers, specifying the starting address of the memory location where the operation should occur and the number of words to be handled. Depending on the architecture on which it will be released, DMAC may possess further features such as interrupt capabilities, different modes

of operation, and so forth.

Operation Modes DMA transfers can occur in three different modalities:

- **Burst mode** In burst mode DMA accesses are performed atomically: once the CPU grants DMA the access to the memory bus, a data transfer starts; it lasts until an entire block of data is sequentially transferred, then DMAC releases the control of the system bus back to the CPU. This is the fastest transfer mode, but it also prevents CPU to access the memory for long periods of time.
- **Cycle stealing mode** The cycle stealing mode is a good compromise between transfer speed and CPU memory access time. This transfer mode should be used in those cases where the CPU cannot be inactive for the whole length of a burst mode transfer. The difference with respect to the latter is that CPU gains back control of the bus after one byte of data has been transferred. Hence block transfer proceeds byte-per-byte, with the DMAC having to request the access to the system bus each time.
- **Transparent mode** Transparent mode is the slowest operation mode, yet the most efficient mode in terms of overall system performance. In transparent mode DMAC transfers data only when the CPU is not using the system bus; the primary advantage of this mode is that CPU never stops waiting for the system bus. The disadvantage is that DMA transfers takes longer times to complete, besides the hardware needs to determine when CPU is not using the system bus; in some cases this can be complex.

2.4.2 DMA Interface for OpenMSP430

The openMSP430 microprocessor offers a DMA interface that acts as gateway to the whole memory of the system.

Memory in the openMSP430 architecture In openMSP430 architecture the memory is seen as a single block, composed by program memory, data memory and peripheral space arranged sequentially as shown in Figure 2.5. Each physical memory location is a 16 bits word, whereas logical memory words are 8 bits long; hence logical memory is perceived as double the dimension of the physical memory. When accessing a logical memory location through a logical address, physical address needs first to be retrieved: this operation is automatically done in the memory backbone of the system, where the logical address is compared with the starting and ending points of logical memory regions (program, data or peripheral

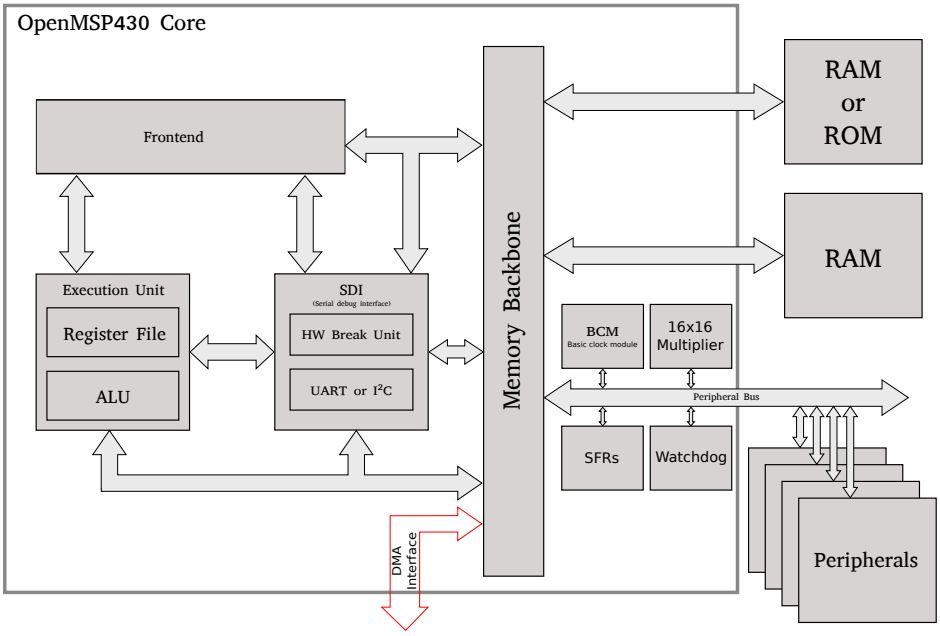

Figure 2.4: Overview of the openMSP430 core internals, with DMA support [18]. Memory can be accessed both through the normal system bus, along with DMA interface.

space), in order to find the correct one to access to. After that, the starting address of the resulting region is subtracted from the logical address $LOGICAL' = LOGICAL - START\_POINT$. Finally, the physical address is obtained with a simple bitwise right shift $PHYSICAL = LOGICAL'/2$.

2.4.2.1 DMA Interface - Signals

Here a list of all the signals of the DMA interface is presented in Table 2.3; each of these signals takes part in a DMA transfer and it's driven either by the core or by the DMAC. Description of the signal is also given. Notice that the DMA interface of the core is meant to be connected to a general DMA arbiter, as DMA controllers, bootloaders, and so forth. For this reason in this section the term DMA Master will be use, to keep the discussion as generic as possible; keep in mind that the DMA Master of the Sancus architecture is a DMA Controller.

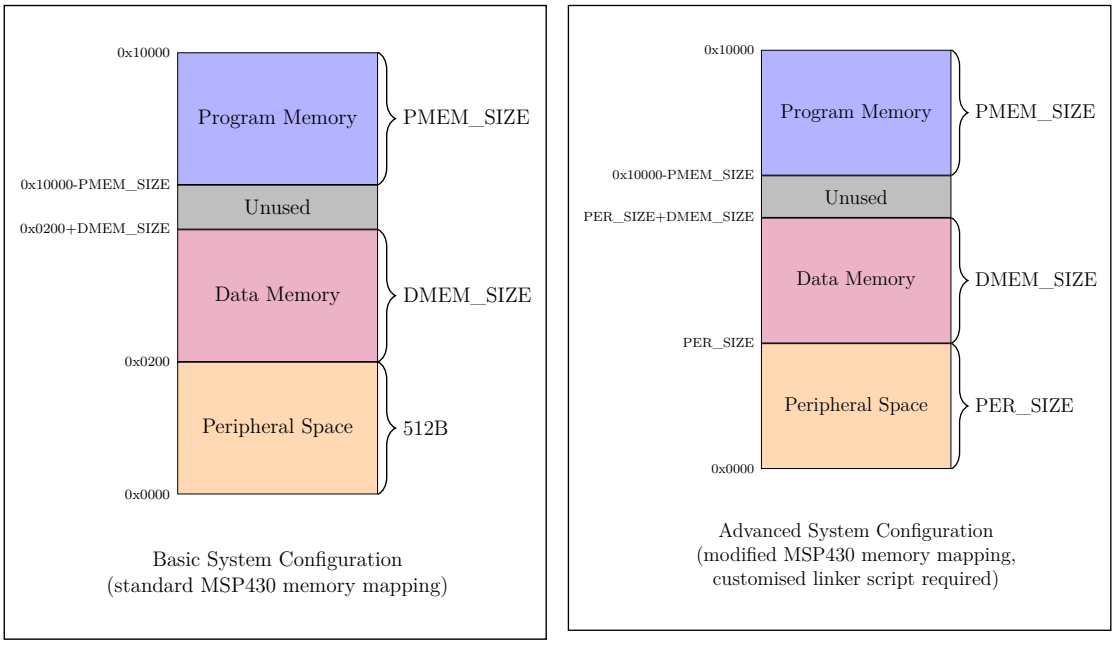

Figure 2.5: Organisation of the memory in the openMSP430 core [18]. Memory is seen as a single memory block, with program memory, data memory and peripheral space arranged sequentially. Each physical memory location is of 16 bits, whereas logical memory words are 8 bits long, hence the logical memory is perceived as twice as the physical memory. A more advanced system configuration is available, allowing to further extend the peripheral space; this would result in a shift of the start and end points of data memory (image on the right).

Table 2.3: List of all the signals of the DMA interface. A brief description of the signals composing the DMA interface, along with their I/O direction, is reported. Notice that the DMA Master can be any device capable of correctly driving the DMA interface signals such as bootloaders, DMA controllers, Memory-BIST or any other hardware unit requiring direct read/write access to the CPU memory space. Source [18].

Signal Name	Description	I/0 Direction
MCLK	It's the system clock that times all DMA transfers.	openMSP430 → DMA Master
PUC_RST	It's the system reset signal and it is active HIGH. This signal is used to reset the system, including the DMA master.	openMSP430 → DMA Master

DMA_DOUT[15:0]	The read data bus is used to transfer data from the openMSP430 memory to the DMA master. It needs to stay valid until transfer is complete, as signalled by DMA_READY.	openMSP430 → DMA Master
DMA_READY	This signal flags a transfer complete when HIGH.	openMSP430 → DMA Master
DMA_RESP	The response signal it's driven HIGH on transfer errors.	openMSP430 → DMA Master
DMA_ADDR[15:1]	Logical address of the 16-bit word currently accessed by the interface.	DMA Master → openMSP430
DMA_DIN[15:0]	It's the write data bus used to transfer data from the DMA master to the system memory. It needs to stay valid until transfer is complete, as signalled by DMA_READY.	DMA Master → openMSP430
DMA_EN	The enable signals indicates that the current DMA transfer is active. It needs to be driven HIGH until completion of the transfer, as signalled by DMA_READY.	DMA Master → openMSP430
DMA_WE[1:0]	Each of the two bits indicate a transfer on the selected byte, when HIGH, and a read transfer when LOW.	DMA Master → openMSP430
DMA_PRIORITY	When this signal is HIGH the DMA master gains high priority on the CPU, thus unlocking *burst mode* accesses. When LOW the DMA Master accesses the system memory in transparent mode of operation.	DMA Master → openMSP430

2.4.2.2 DMA Interface - Protocol

A DMA transfer starts when enable signals goes HIGH. At that moment the *DMA_ADDR* and the control signals for the required operation, read or write, need to be asserted and to stay valid until the end of the transfer is flagged by the *DMA_READY* signal. When this latter is HIGH it means that the core has successfully handled the required operations; however when *DMA_READY* is LOW it inserts wait states into the transfer. The DMA Master needs to acknowledge the wait state and to keep the signal unchanged until *DMA_READY* is driven HIGH

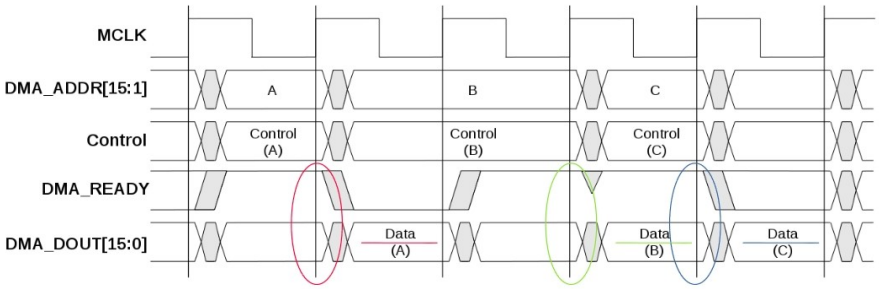

Figure 2.6: Timing diagram of a read operation with wait state on the openMSP430 DMA interface. The read data DMA_DOUT are available on the next clock cycle after the DMA_READY signal has been asserted by the core internal logic. If the signal is LOW, a wait state is inserted in the transfer, as it happens for the case of the datum B: in this case, the DMA master keeps the DMA address, as well as with the control signals, unchanged until the ready signals is asserted again [18].

again, signalling the completion of the transfer. Transfer errors are signalled by having *DMA_RESP* driven HIGH; errors are generated if the transfer address lays between the program and data memories, where nothing is mapped (for example between Program and Data memory, as shown in Figure 2.5). Notice that ERROR warnings have never wait states. A separate description is required for both the read and write operations, as they are differently handled in the protocol.

A low value of *DMA_WE* together with an active *DMA_EN* triggers a read operation on the selected byte: *DMA_WE[0]* activates a read on the lower byte, *DMA_WE[1]* on the upper byte. The DMA Master drives the address and the control signals, and waits for openMSP430 to sampled them. When the system

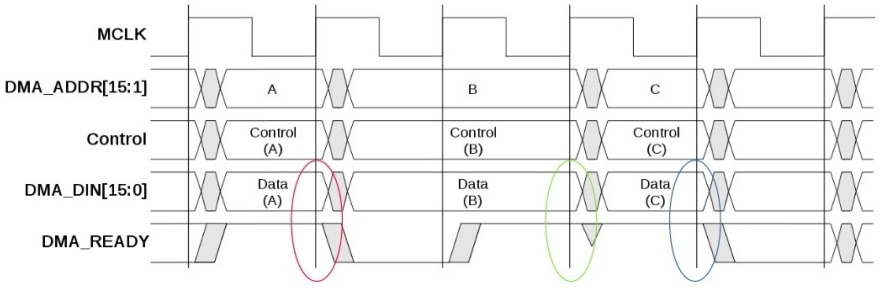

Figure 2.7: Timing diagram of a write operation with wait state on the openMSP430 DMA interface. DMA_READY signal is asserted on successfull sampling of the incoming data, as it happens for data A and C. In case of wait states, the DMA_READY signal is driven low and the DMA master waits until the trasnfer is completed. This is shown for data B, as one wait state is insterted before the data is finally sampled [18].

core does so, it raises *DMA_READY* high. Data will be available on the next clock cycle after *DMA_READY* has been risen, as shown in Figure 2.6.

An high value of *DMA_WE* together with an active *DMA_EN* triggers a write operation on the selected byte: *DMA_WE[0]* activates a write on the lower byte, *DMA_WE[1]* on the upper byte. During a write operation the data on the write bus *DMA_DIN* is stored into the system memory, at the address specified by the DMA Master. This latter keeps those signals valid until the openMSP430 asserts *DMA_READY*, signal that the transfer is completed. The timing diagram of the write transfer can be found in Figure 2.7.

Chapter 3

Problem Statement

3.1 DMA on Protected Module Architectures

DMA is generally not supported in PMAs. The reason is that PC based memory access control enforced on PMAs (section 2.2) can only check and validate memory accesses coming from the CPU memory bus. However, if a DMA interface is provided to PMAs, a secondary direct channel to the memory is opened. In this way, a peripheral with full DMA capabilities can access every mapped memory location, including private SMs code and data sections, thus completely breaking isolation and confidentiality guarantees for the system. Lightweight PMAs, such as Sancus or TrustLite, as well as high-end solution, such as SecureBlue++ [7] do not include DMA on their systems at all. Whether this may seem a trivial solution, it has the advantage of being simple and not adding any complexity to the architecture. This can be crucial, especially on low-end embedded PMAs were the resource constraint is the main limiting design parameter. A discussion on possible solutions to support DMA on PMAs without affecting their security guarantees is deferred to chapter 4 *"Design and Discussion"*.
Others, like Iso-X [17] or Intel SGX [13] explicitly prevent any access to the protected memory. In these architectures, the equivalent of modules protected sections are stored in a specific range of the memory space. Thus, the protection mechanism consists in denying every DMA accesses to those specific regions.
In Intel SGX, for example, the Enclave Page Cache (EPC) provides the protected memory region for enclaves in the machine. The Enclave Page Cache Map (EPCM) is the security meta-data attached to each EPC page, and contains the information needed by the hardware to protect the enclave memory accesses. All these entries are stored in the Processor Reserved Memory, which is a a contiguous range of DRAM that cannot be accessed by system software or peripherals. An image of the memory hierarchy in use on Intel SGX is provided in Figure 3.1.

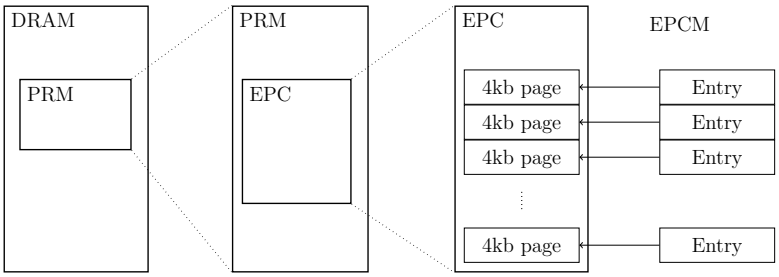

Figure 3.1: Detail of Processor Reserved Memory (PRM), together with Enclave Page Cache (EPC) in Intel SGX. The PRM is a contiguous range of DRAM that cannot be accessed by system software or peripherals [13].

3.2 Exploitation of Naive DMA Support

A simple example of how security guarantees are invalidated when DMA support is naively implemented on Sancus 2.0. If no memory access control is enforced on the DMA bus, it can access to the whole system memory, including protected sections. An attacker can exploit this vulnerability to leak secret data or inject malicious code into the module.
Listing 3.1 shows a simple Sancus software module *"Hello"*: the macros `SM_ENTRY(name)`, `SM_FUNC(name)` and `SM_DATA(name)` are respectively used to annotate modules entry points, functions or data. In brief, they take as input the name of the intended module, and attach some attributes[1] to the entity they are associated with (data, function or entry point), so that the Sancus compiler can include those in the module protected sections. `DECLARE_SM(name, vendor_id)`, instead, is used to declare a Sancus module.

Listing 3.1: Code snippet showing a SM named *"Hello"*. Macros are used to annotate the code so that the Sancus compiler can correctly enclose SM functions and data respectively in the SM text and data sections.

```
/* ======== HELLO WORLD SM ======== */
DECLARE_SM(hello, 0x1234);
#define HELLO_SECRET_CST    0xC1A0

int       SM_DATA(hello) *hello_secret;
int const SM_DATA(hello)  hello_const = HELLO_SECRET_CST;
```

[1]The attribute mechanism allows a developer to attach extra information (metadata) to language entities with a generalized syntax, instead of introducing new syntactic constructs or keywords for each feature. This information is intended to be used by the compiler, improving the quality of diagnostics produced by an implementation or specifying platform-specific behaviour [5].

```
void SM_FUNC(hello) hello_init(void)
{
    /* Confidential loading guarantees secrecy of constant in
       text section. */
    hello_secret = (int*)(hello.data_section_start+OFFSET);
    *hello_secret = hello_const;
    ASSERT(*hello_secret == HELLO_SECRET_CST);
}

void SM_ENTRY(hello) hello_greet(void)
{
    hello_init();
    pr_info2("Hi from SM with ID %d, called by %d\n",
       sancus_get_self_id(), sancus_get_caller_id());
    pr_info2("Internally accessing to my secret: %.4x at addr.:
       0x%.4x \n",*hello_secret, hello_secret);
}

void SM_ENTRY(hello) hello_disable(void)
{
    sancus_disable(exit_success);
}
```

3.2.1 Leak Secret Data

In the example provided the value of a secret constant is stored in the data section, at a known offset, for the sake of simplicity: in this way the DMA attack is carried in one shot. Though in a real-case scenario it's unlikely for the attacker to know the exactly location of the secret data, it is reasonable to assume that some analysis are carried on on the module to be broken, thus the example it's still valid. Finally, it is to be noticed that the values of start and end addresses of SMs text and data sections can be read from unprotected code.

```
int main()
{
    uint16_t start_dma;
    uint16_t disclosed_secret;

    msp430_io_init();
    sancus_enable(&hello); //Enable memory protection

    hello_greet(); //Module's function, automatically accessed
       from the unprotected code through the entry point


        /* ============== USING DMA ================ */
```

```
    start_dma = (uint16_t)(hello.data_section_start+OFFSET);
    pr_info("DMA illegal access to the secret data from
       unprotected code \n");

    //void dma_read (start_addr, num_of_words, *save_data)
    dma_read(start_dma, 1, &disclosed_secret);
    pr_info2("Hello secret is: 0x%.4x at address: 0x%.4x \n",
       disclosed_secret, start_dma);
        /* ======================================== */
}
```

Listing 3.2: Main.c function. SM *"Hello"* secret is leaked through a DMA read after the memory protection has been enabled.

Listing 3.3: Output of the terminal in which the attack example was launched. After the DMA read operation is called from the main.c, the secret data *"C1A0"* is read and stored in `disclosed_secret` (Listing 3.2). Then, its value is printed from the unprotected main.c, meaning that the DMA successfully leaked the secret. None of these operation raised an exception.

```
New SM config: 7130 7304 029c 03a6, 1
Vendor key: 4078d505d82099ba
.....................................................................

SM key: b0c4cc0fb9ce3806
SM hello with ID 1 enabled      : 0x7130 0x7304 0x029c 0x03a6
[main.c] Hi from SM with ID 1, called by 0
[main.c] Internally accessing to my secret: 0xC1A0 at address: 0
   x02a3
[main.c] DMA illegal access to "Hello" secret from unprotected
   code

[main.c] Hello secret is: 0xC1A0 at address: 0x02a3
SM disabled
[main.c] SM disabled; all done!
 ================================================
|               SIMULATION PASSED                |
 ================================================
```

After entering the main function (Listing 3.2), the memory protection for the "Hello" module is enabled. Then, the SM `hello_greet()` is called, which initialises the secret data and sends a greeting to the reader.
The execution continues by configuring a DMA read operation from the main.c. The DMA read function inputs are: (1) the starting address, which is set with the start of data section shifted of the know offset, so that it directly points to

the desired secret data; (2) the number of words to be read, set equal to one in this case; (3) the address of an unprotected memory location where to store the read data. Once the DMA operation has completed, the secret value *0xC1A0* is leaked and stored and printed from code in the unprotected memory. The output produced by the main.c is shown in (Listing 3.3).
As a final remark, Listing 3.4 shows what happens when the same code from Listing 3.1 and Listing 3.2 is run on Sancus with memory access control enforced on DMA accesses: a violation is internally raised and the content of the secret data is not leaked. An interrupt request can be set so that more specific countermeasure can be taken when a DMA violation is detected; an example of DMA ISR is provided in Listing 3.5.

```
New SM config: 7130 7304 029c 03a6, 1
Vendor key: 4078d505d82099ba
.....................................................................

SM key: b0c4cc0fb9ce3806
SM hello with ID 1 enabled      : 0x7130 0x7304 0x029c 0x03a6
[main.c] Hi from SM with ID 1, called by 0
[main.c] Internally accessing to my secret: 0xC1A0 at address: 0
   x02a3
[main.c] DMA illegal access to "Hello" secret from unprotected
   code

[main.c] Hello secret is: 0x0000 at address: 0x02a3
SM disabled
[main.c] SM disabled; all done!
 ================================================
|                SIMULATION PASSED                |
 ================================================
```

Listing 3.4: Output of the terminal for the same example of Listing 3.1 and Listing 3.2, running on Sancus with memory access control enforced on DMA accesses.

Listing 3.5: Detail of ISR to handle DMA violations. The attribute *"interrupt"* identifies the function as ISR.

```
__attribute__((interrupt(DMA_IRQ)))
void dma_violation_isr(void)
{
    puts("\t--> DMA VIOLATION IRQ; exiting...\n");
    EXIT();

    pr_info("should never reach here..");
    while(1);
}
```

3.2.2 Inject Malicious Data or Code

Another possible attack implies the injection of malicious code or data into SMs protected sections. The SM considered is still the *"Hello"* module from the previous example (Listing 3.1). The only difference is that the SM `hello_init()` function is now explicitly called only once right after the protection mechanism has been enabled, and not every time the `hello_greet()` is entered. To this extent, its attribute is modified from SM_FUNC to SM_ENTRY. This is necessary, in order to prevent that the injected malicious message is simply overwritten before being printed by the greeting function. Listing 3.6 shows the main.c, as framework for the attack.

Listing 3.6: Main.c function for the write attack. A malicious data is written into the SM *"Hello"* data section, through a DMA operation.

```
int main()
{
    uint16_t start_dma;
        uint16_t data_to_send;

    msp430_io_init();
    sancus_enable(&hello); //Enable memory protection

    hello_init();  //Now defined as: void SM_ENTRY(hello)
       hello_init(void)
    hello_greet(); //Module's function, automatically accessed
       from the unprotected code through the entry point
        /* ============== USING DMA ================ */
    start_dma = (uint16_t)(hello.data_section_start+OFFSET);
    data_to_send = 0xBEEF;
```

```
        puts("DMA injecting external data into SM data section \
            n");

        //void dma_write(start_addr, num_of_words, *data_to_send
            );
        dma_write(start_dma, 1, &data_to_send);
        /* ======================================== */

        hello_greet();
}
```

After the greeting message has printed, to show that initialization has correctly occurred, the real attack can start, by configuring the DMA write function. It takes three inputs: (1) a starting address pointing to the initial memory location where the external data are going to be written; (2) the number of words involved in the operation; (3) the first memory location where data resides. As for the previous example, the writing address is still set to a known offset in the data section. Then, the malicious greeting message `0xBEEF` is injected and the execution of the main function proceeds by calling the SM `hello_greet()` again, to show that a tempering occurred (Listing 3.7).

```
New SM config: 7124 72ea 029c 03a6, 1
Vendor key: 4078d505d82099ba
.....................................................................

SM key: ed151c6be5efda74
SM hello with ID 1 enabled      : 0x7124 0x72ea 0x029c 0x03a6
[main.c] Hi from SM with ID 1, called by 0
[main.c] Internally accessing to my secret: 0xC1A0 at address: 0
   x02a3
[main.c] DMA injecting external data into SM data section

[main.c] Hi from SM with ID 1, called by 0
[Hello.c] Internally accessing to my secret: 0xBEEF at address: 0
   x02a3
SM disabled
[main.c] SM disabled; all done!
 ================================================
|               SIMULATION PASSED                |
 ================================================
```

Listing 3.7: Output of the terminal for the write attack.

Chapter 4

Design and Discussion

The aim of this chapter is to present the design steps for extending Direct Memory Access (DMA) support to Sancus. The security objectives, pursued throughout the whole design process, start the discussion in section 4.1. Then, the attacker model is presented (section 4.2), and an general overview of the system is provided (section 4.3). Afterwards, the threats of direct DMA integration on Sancus are analysed (section 4.4), and possible solutions outlined, each presented with its pros and cons (section 4.5).

4.1 Security Objectives

The main design challenge of the thesis is to provide the Sancus architecture with DMA support, without affecting its security guarantees. In particular software modules protected sections have to remain inaccessible from code executing outside the module itself. This means that accesses through the DMA interface should not affect the integrity of the software modules (from now on SMs).

Direct implementation of the DMA support does not comply with the security objective just stated: if DMA is provided, every memory mapped location can be access, thus invalidating SMs isolation property.[1] Therefore, a more sophisticated approach has to be pursued. First idea is to enforce a program counter-based memory access control on DMA accesses, as explained in subsection 4.5.2; however, this opens to memory right escalation attacks. The failure of this solution is mainly due to the flawed idea of using CPU-related entity - the program counter (PC) - to validate DMA accesses, which are, by definition, happening independently of the CPU.

A different approach is the key to fulfil the security guarantees stated in the start of the section: DMA is entirely excluded from protected memory regions, in

[1] A detailed review of software modules security guarantees can be found in subsection 2.3.2.

order to keep modules integrity and confidentiality untouched. Subsection 4.5.3 describes it, together with its pros and cons.

The solutions presented so far aim to keep security guarantees unaltered. Nevertheless it is possible to relax some constraints on integrity and confidentiality, and trade security for SMs functionalities, in a controlled way. This is proposed in subsection 4.5.4, in which each module is allowed to define a specific memory subset inside its data section to be disclosed to DMA peripherals with DMA capabilities, if needed.

A final mention is for side channels attacks as threat for the overall security of the system. By extending the relying architecture with Direct Memory Access support, the side channel attack base is extended, too. The discussion about this topic is deferred to section 4.7.

4.2 Attacker Model

Throughout the design of the DMA support and of the strategies to prevent its malicious exploitation, attackers are generally considered to possess the same capabilities as in Sancus 2.0 attacker model [39], unless differently stated. Specifically, the followings are assumed:

- Attackers can manipulate all the software on the nodes. In particular, they can act as a software providers and deploy malicious modules on the node, tamper with the operating system, or even install a completely new one.
- Attackers can control the communication network that is used by the software providers and nodes to communicate between each other. Independently of the security of the communication channel, which is out of scope, they can sniff the network, modify traffic, or mount man-in-the-middle attacks.
- Attackers cannot break cryptographic primitives, but they can perform protocol-level attacks.[2]
- Attackers do not have physical access to the hardware of the system, which means they cannot place probes on the memory bus nor disconnect components, at anytime. However, attackers are allowed to plug-in their own peripherals or to substitute the DMA controller provided by default with their own version of it. These modifications need to be carried out before the system is started, as any further alteration at runtime is not considered in this model. Notice that

[2]Protocol-level attack is defined as the exploitation of specific feature or implementation bug of some protocol installed at the victim for consuming huge amount of its resources.

security guarantees from section 4.1 must still hold even in this eventuality. This is a new property with respect to. the attacker model of Sancus 2.0 [39]. Figure 4.1 shows an overview of the architecture. Here the Trusted Computing Base (TCB) is highlighted, too: it is defined as the set of the hardware components[3] which are considered critical for the security of the whole system. The TCB is designed so that, even in case of tampering with other parts of the system, the device cannot misbehave [30]. The attacker model considers the TCB inaccessible for the attackers, at anytime. Otherwise, security objectives of section 4.1 are not guaranteed. On the contrary, no assumption is made for the components outside the TCB, including the DMA controller and the peripherals.

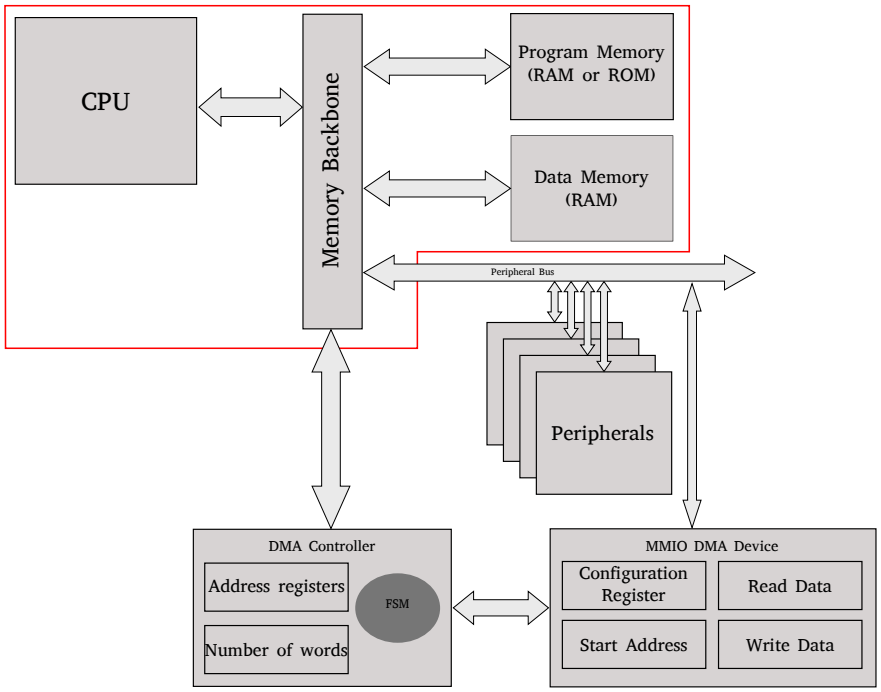

Figure 4.1: Overview of the system architecture, showing the DMA controller and a peripheral with DMA capabilities, connected to it. Everything related to the DMA or peripherals is outside the TCB, whose boundaries extends to the CPU, memory backbone and the system memories. In fact, no assumption is needed on DMA controller nor devices trustability.

[3]Generally speaking, the Trusted Computing Bases can include both hardware and software components that guarantees the security of a system. In Sancus, the TCB is hardware-only, as the software is vulnerable to manipulations of the attacker, hence cannot be trusted.

4.3 Overview

Sancus 2.0 [39] is a secure architecture which relies on strong security guarantees, listed in subsection 2.3.2. In summary, SMs are binary files composed of a text- and data- sections: after modules are deployed and memory protection is enabled, the contents of the two sections are guaranteed to be inaccessible from code executing outside the SMs, fulfilling the so called isolation property. Furthermore, secure communication among modules or with a remote party is achieved through specific instructions which verify that the intended module *SM*, running on node *N* on behalf of software provider *SP*, has been called (see section 2.3 for a deeper explanation).

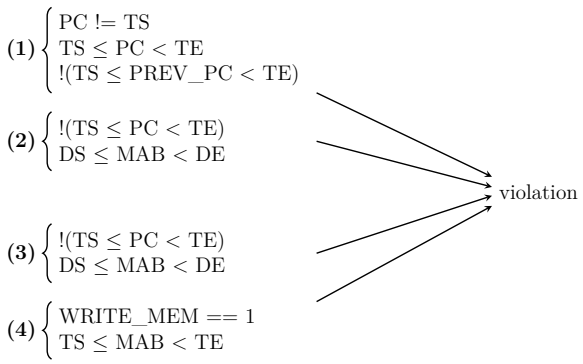

Figure 4.2: Representation of the Memory Access Logic (MAL), enforcing program counter-based access control rule. In this high level representation, the MAB is validated on the current and previous value of the program counter. TS and TE stand for Text section Start and End addresses, whereas DS and DE are the Data section Start and End addresses.

Sancus does not natively support a DMA interface, hence all the memory accesses pass through the CPU, on the memory access bus (MAB). Thus, it is sufficient to enforce a program counter-based memory access control on the MAB in order to guarantee SMs' integrity and confidentiality. This is achieved by the memory access logic (MAL), whose high-level representation is shown in Figure 4.2. A violation is raised every time one of the four conditions is satisfied: (1) if the program counter is addressing to the text section of a module without using its the entry point; (2) if the MAB is pointing to the data section of a module, while the PC resides outside that module; (3) if the MAB is pointing to the text section of a module, while the PC resides outside that module; (4) if a write operation is being carried on into the text section of a module.

4.4 Impact of the DMA on Sancus Security Properties

By providing Sancus with a DMA interface, a secondary channel to the memory is opened. In this way, a peripheral with full DMA capabilities can access every mapped memory location, completely breaking isolation and confidentiality guarantees for software modules. In fact, PC-based memory access control is not enforced on DMA accesses, which directly address the system memory. This opens new breaches in the system, as the content of a software module, both text- and data- sections, becomes fully accessible to an attacker with DMA capabilities, without even raising an exception in the MAL.
Here follows a description of how DMA affects Sancus security guarantees: the focus is on module isolation property, as all the other security guarantees base on that. In fact, if module content is going to be illegally accessed, then any effort to secure modules communication within and outside the core becomes secondary, as the attacker will always be able to retrieve secrets by directly reading into the protected memory regions of the implicated modules.

- **Software module isolation:** An attacker with DMA capabilities can read or write every memory location, including the protected section of a module. Thus, module isolation entirely reneges. Any illegal read operation leaves no trace on the module, as it does not modify any part of the two sections; thus it cannot be detected neither with subsequent remote attestations by the *SP*. On the other hand, any modification due to an illegal write operation can be detected by the *SP* with successive remoter attestations, as they affect the module identity *SM*.

- **Remote attestation:** Remote attestation strongly relies on SM's isolation. The attestation mechanism, as described in subsection 2.3.2, leverages on the assumption that a module cannot be modified after the protection has been enabled. This is true if Sancus does not support DMA, since in such architecture a module can solely be tampered with before the protection instruction is executed. After that moment, the memory access control enforced on SM assures that no further modifications take place, hence that the SM's key $K_{N,SP,SM}$ is uniquely associated with the untampered module *SM*, deployed from the software provider *SP*, running on the node *N*.
  Therefore the module key $K_{N,SP,SM}$, which is saved in the Protected Storage Area (PSA) of the processor, can be used to generate a MAC of a nonce sent by the SP. This latter computes its own version of the MAC, and compares with the once received from the SM: if the two coincide, then the attestation succeeded.

  If Sancus is provided with a DMA interface, modules content can be accessed (both read or written) at anytime during the execution, thus invalidating the

isolation property. Recalling that the module key $K_{N,SP,SM}$ is computed only once when the `protect` instruction is called,[4] it can no longer be consider a sufficient assurance of module integrity. An attacker, in fact, could tamper with the SM after the protection is enabled, without causing any change in the module key, hence tricking the attestation mechanism.

Finally, a Sancus feature has to mentioned: by having node and modules keys saved in the PSA of the processor, they are never leaked. A malicious DMA operations cannot access them, as the PSA it is not mapped into the system memory, nor a malicious SM ad-hoc programmed, since the keys are only indirectly accessed by software through processor instructions: node key K_N is involved in the creation of $K_{N,SP,SM}$ by the `protect` instruction, whereas $K_{N,SP,SM}$ is used in `encrypt` or `decrypt` functions. This is a peculiarity of the Sancus architecture in contrast to other solutions which leak the key, as it happens, for example, with Intel SGX in Foreshadow attack [9].

- **Secure communication:** The secure communication between a software provider *SP* and a module *SM* also leverages on SM isolation property and on the $K_{N,SP,SM}$ key, as a proof of integrity for an intended module.
  When a SP wants to establish a secure communication with a SM, it sends a nonce N_0 and possibly some data I to the node N; those are passed to the SM function to be called, by untrusted code. Then, SM output O is encrypted by the `C,T = encrypt(`$K_{N,SP,SM}$`,O,` N_0`||I)` instruction, and the resulting ciphertext and tag are sent to the SP on the untrusted network. By comparing its own version of the tag with the one received from the SM it can be sure that the incoming data have been produced by SM, running on node N, on the given input I.
  As for the remote attestation of above, the module key $K_{N,SP,SM}$ can no longer be consider a sufficient assurance of module integrity, as it is computed only once after protection is enabled, thus an attacker could always tamper with the SM from that moment on.

- **Secure linking:** DMA capabilities invalidates this property, too. In fact, modules are attested only once in secure linking; if the attestation succeeds, ID of the module is stored, to be used in place of subsequent attestations. Therefore any modification of the module carried out after the first attestation is not detected, and module ID it is not revoked. The extent of this vulnerability is relevant, as an attacker can exploit a modified module to interact with all the modules it has access to.

[4] A detailed explanation of `protect` instruction can be found in section 2.3.2 *"Software module isolation"*.

- **Confidential loading:** When a module is deployed on Sancus, its text section can be accessed until the the protection has been enabled. In the non DMA-supporting Sancus, this is the only way the text section could be illegally accessed, as long as module isolation holds. In order to gain confidentiality guarantees for SMs, a SP can send an encrypted version of the text section of the module, which is then decrypted in place using the $K_{N,SP}$.

  DMA capabilities completely break this property. In fact, even if confidential deployment is correctly completed, it is not possible to keep module content secret anymore, as an attacker can access it through a DMA operation anytime.

- **Hardware breach confinement:** This property is not affected by the presence of DMA interface. In fact, all the keys used in Sancus are securely stored in the PSA of the core (Figure 2.1), which cannot be accessed neither by hardware nor software. Therefore, keys are never leaked neither in case of illegal DMA operations, as the PSA it is not mapped into the system memory, nor in case of a malicious software module ad-hoc re-programmed, since the keys are only indirectly accessed by software through processor instructions. Hence, in the event a node is compromised, an attacker cannot retrieve any information about other nodes and the breach stays confined.
  Even if the key is never leaked, an attacker has still the possibility of entirely rewriting the text section of a module, de facto gaining full control on the compromised module and making it a Trojan horse [27]. Some may object that this scenario is equivalent to the disclosure of $K_{N,SP,SM}$, as the attacker can now impersonate that module. However an important difference still exists: even by gaining control on a SM, attacker computational capabilities are always confined to node where the SM is running. On the contrary, if the key leaked, it would be possible to run Sancus cryptographic functions on any external processor provided with the $K_{N,SP,SM}$.

4.5 Protection of System Memory from DMA Attacks

The disastrous outcomes of allowing DMA peripherals to have full memory access strengthen the need to introduce memory protection strategies to prevent malicious DMA operations. This section goes through some solutions to this problematic; some of them are from real case systems as TrustLite [26], SMART [14] or Intel SGX [13], whereas others have been designed specifically for Sancus. A pros and cons analysis is provided for each of them.

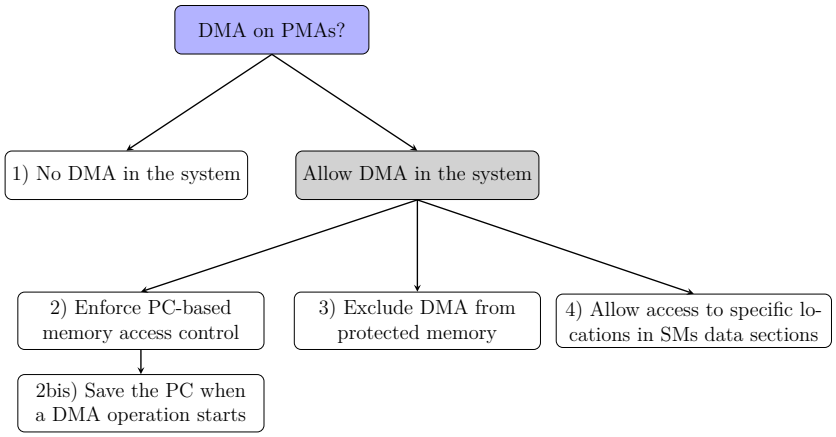

Figure 4.3: A recapitulatory tree diagram of the explored ideas to include DMA on PMAs. Among these, only the last leaves, numbers 3 and 4, propose an actual solution to the problem.

4.5.1 No DMA in the System

Even if it may seem a trivial solution, this is the best trade-off for lightweight systems. Usually these architectures are developed on low-end microcontrollers where the cost in terms of resources, such as chip surface, chip complexity, power and performance, is the main limiting design parameter. In such systems it can be a reasonable choice to sacrifice DMA capabilities in order to keep security guarantees, without adding any complexity to the architecture. This strategy is pursued in some known PMAs like, for example, SMART and TrustLite.

Pros	Cons
• Very simple and cheap choice	• Does not provide any DMA capability

4.5.2 Enforce MAL on DMA Accesses

An initial idea is to extend PC-based memory access control to the DMA in order to prevent any illegal access to protected memory regions. In this scenario hardware modifications need to be carried out: the DMA bus, which usually is directly connected to the memory, now has to be rerouted in the memory access control circuitry that validates the execution unit MAB; the previous MAL of Figure 4.2 is then updated into a DMA-including version, shown in Figure 4.4, where the memory access control is enforced either on the MAB or on the DMA_ADDR, depending on whether or not the DMA is enabled.

Unfortunately, this opens a breach for memory escalation exploitation, as the

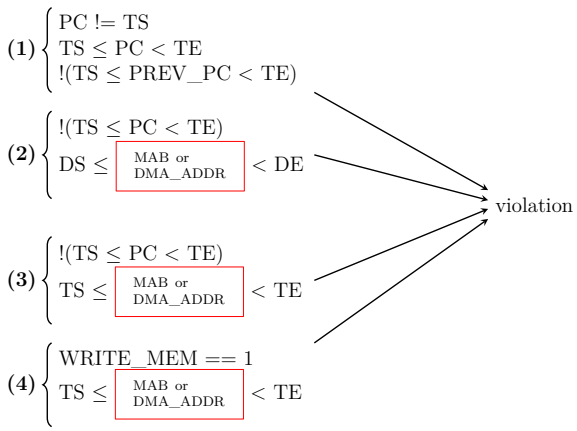

Figure 4.4: Modifications to the Memory Access Logic (MAL) are here shown. The circuitry now enforces PC-based access control on the DMA address: both the MAB and the DMA_ADDR are validated on the current and previous value of the program counter; the selection between the two depends on whether or not the DMA is enabled. TS and TE stand for text section start and end addresses, whereas DS and DE are the data section ones.

program counter is free to vary during a DMA operation. A possible attack scenario has been theorised, with a malicious DMA operation that starts while the PC is accessing to unprotected memory; when the PC enters in a protected memory space, any further DMA access is authorised to access protected data. Hence, an adversary with DMA capabilities could both read and write into modules sections, without even raising a violation exception.
An idea to fix this vulnerability is to keep validating DMA accesses with the same memory permissions of when the operation started. In order to do so, it's necessary to store the value of the PC at the start of an operation, and keep it until its end. Whereas this prevents right memory escalation while an operation is happening, it does not entirely secure the system. Actually, an attacker can still start a DMA operation right when the PC is in protected memory; by allowing the saving of the PC, it is provided with the possibility of keeping accessing the protected region even when the current PC is outside of it. This opens to an even more dangerous scenario, endangering modules integrity and confidentiality until the operation ends.

Both of these solutions try to enhance the security guarantees on DMA accesses, but inevitably lead to vulnerabilities in the system. As said in section 4.1, their failure is mainly due to the use of a CPU-related entity - the program counter (PC) - to validate DMA accesses, which are, by definition, happening independently of the CPU.

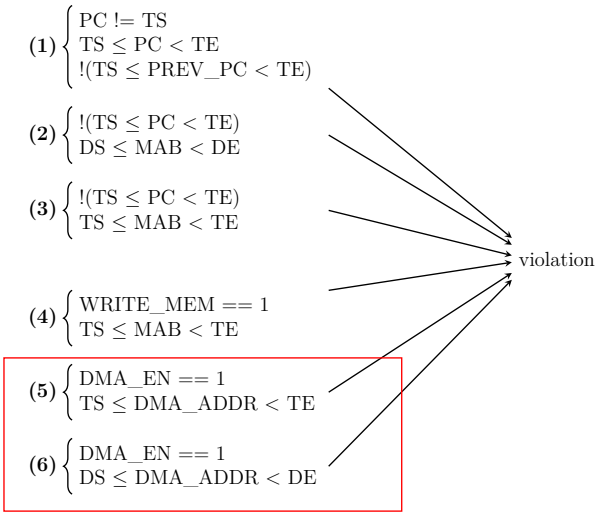

Figure 4.5: Representation of Memory Access Logic (MAL) that prevents any DMA access to protected memory. Every time the DMA address poits to a protected memory location, while the DMA is enabled, a violation is raised. TS and TE stand for text section start and end addresses, whereas DS and DE are the data section ones.

Pros	Cons
• Allows DMA in the system • Expands the already instantiated MAL circuitry with minimal hardware additions	• Fails in protecting the system

4.5.3 Exclude DMA from Protected Memory

A different approach is to totally prevent DMA accesses to protected memory regions. Even if it may seems too drastic, its effectiveness is undeniable. Intel SGX pursues this approach in protecting PRM memory, as explained in [48, 13], in which the contents of enclaves[5] and the associated data are stored. In Intel SGX the reserved memory is a continuous subset of DRAM;[6] hence, the protection mechanism simply consists in denying access to that specific range to software or peripherals, including DMA.

On Sancus, instead, software modules are instantiated across the whole memory. The only way to prevent DMA accesses to SMs text and data section is to modify

[5]Intel SGX enclaves can be considered equivalent to Sancus software modules.

[6]A representation of Intel SGX memory structure can be found in Figure 3.1 from section 3.1.

the MAL circuitry so that, every time the DMA address points to a protected memory location while the DMA is enabled, a violation signal rises and the operation stops. An high-level view of the resulting MAL is reported in Figure 4.5. Such solution trades modules DMA capabilities for an enhancement in integrity and confidentiality of protected memory regions.

Pros	Cons
• Allows DMA in the system, preventing accesses to protected memory (SMs integrity and confidentiality preserved) • Reuse of the already instantiated MAL registers TS, TE, DS and DE • No software overhead or SMs direct intervention required	• Allows DMA operations only involving unprotected memory. Does not really extend SMs functionalities

4.5.4 Allow Access to Specific Locations inside SMs Data Sections

Although it prevents illegal memory accesses, Intel SGX approach might be considered unpractical, as it entirely prevents DMA to the SMs contents. It would be interesting, instead, to provide SMs a way to selectively allow DMA operations on their data sections. On the other hand, it would be also advisable to limit the addressable space for a DMA operation, considering that sensitive data reside in modules data sections, like for example modules runtime stacks.
An idea to extend this functionality is to provide the MAL with two addresses, which identify the DMA Protected Start (DMA_PS) and DMA Protected End (DMA_PE) of a reserved memory block inside the data section of a module. This data section subset can be used, then, to provide DMA access to all the peripherals, in a secure way. In fact, in case of illegal DMA accesses, the outcomes would be confined to that specific memory block. However, it is software developers responsibility to keep that memory region free from sensitive data.
Hence, the proposed solution enhances SMs functionalities, as it provides them a way to rely on DMA capabilities, by relaxing integrity and confidentiality guarantees for a specific subset of the data section. In other words, it proposes to trade security for functionalities, in a controlled way. The value of DMA_PS and DMA_PE addresses must be stored in the PSA, together with the text and data sections start and end addresses (TS - TE and DS - DE), therefore it is to be noticed that the proposed solution comes with an overhead of two register for each software module in the node. Furthermore, the value of DMA_PS and DMA_PE

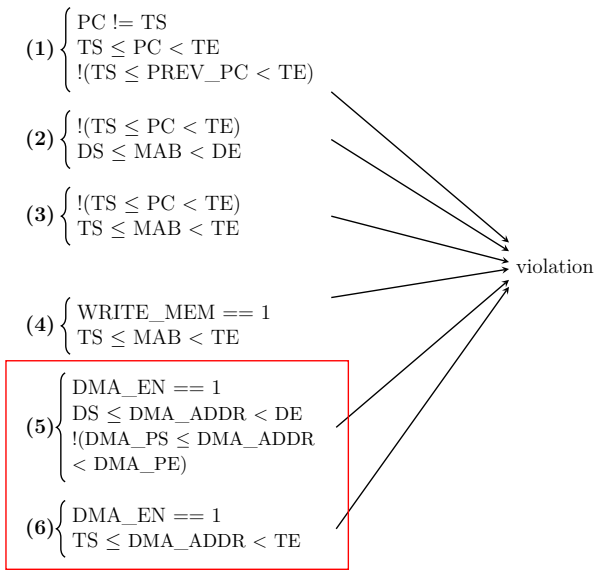

Figure 4.6: Representation of the Memory Access Logic (MAL) that allows DMA access to specific protected memory locations. A violation is raised whether the DMA address lies outside the allowed boundaries, inside modules data section. TS and TE stand for text section start and end addresses, whereas DS and DE are the data section ones; DMA_PS and DMA_PE are the DMA Protected Start and DMA Protected End addresses, and define the data section subset boundaries.

have to be zero set every time a new module is deployed, in order to guarantee its freshness. If not, a threat to the data section integrity and confidentiality appears. In fact, a module not willing to cut out a DMA-allowed space from its own data section, never overwrites the content of DMA_PS and DMA_PE. If those values haven't been set to zero on deployment, the new module would inherit the values of an old one, disclosing a subset of its data section to DMA devices.
Thus, a modification to the processor *protect* instruction of section 2.3.2 is required. Specifically, the new version of the instruction should:

- check that the layout of the new module does not overlap with any existing modules and, if so, continue with the deployment by storing its layout in the PSA of the processor;
- zero set the content of DMA_PS and DMA_PE;
- enable the memory access control on that module;
- create the module key $K_{N,SP,SM}$ basing of the layout informations, and store it in the PSA of the node;

Figure 4.7 shows a schematic of a node implementing the DMA-allowed memory

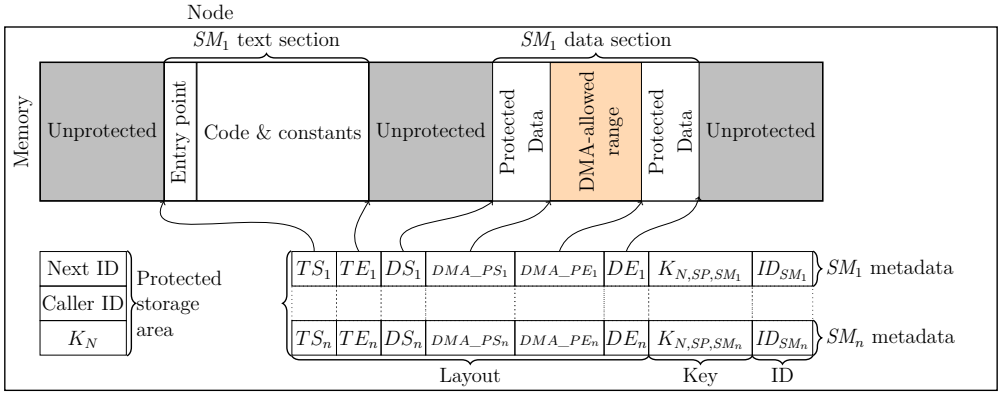

Figure 4.7: Representation of a node with a software module loaded, in the new framework of allowing DMA access to specific locations inside the data section. The content of the processor Protected Storage Area (PSA) is extended with Protected Start (PS) and Protected End (PE) addresses, which identify the DMA-allowed subset inside the data section.

subset. Notice that, in the image DMA_PS and DMA_PE are included in the layout of the module just for the ease of drawing. However, their value is zero set right before the module key $K_{N,SP,SM}$ is created, thus their value would actually have no influence on the layout of the module.

Finally, each module should be able to specify its own DMA-allowed protected space by setting the values of DMA_PS and DMA_PE. Hence, the processor instruction must be expanded with a new instruction *set_dma*, whose task is to check that DMA_PS and DMA_PE are pointing to memory locations inside the data section of the intended modules and, only then, proceed with the saving of those values into the PSA. If it wasn't so, a malicious module could arbitrarily set the size of the protected space of another, thus invalidating software module isolation, again.

The memory access logic circuitry that allows DMA access to specific protected memory locations is here presented in Figure 4.6.

Pros	Cons
• Allows DMA in the system, preventing accesses to protected memory (SMs integrity and confidentiality preserved)	• Register overhead: two extra registers for each SM
• Full configurability of the system: SMs can decide whether or not to provide DMA access to a specific portion of their data sections	• Implicit trustworthiness of all the DMA peripherals as the current system does not provide a way to selectively grant access to a module DMA-region
• Reuse of the already instantiated MAL registers TS, TE, DS and DE	• Extension of the ISA, as modules needs to set the boundaries of their data section subsets through a new processor instruction

4.5.4.1 Reduce the Register Overhead

A different design choice is to set one of the two ends and let the other one vary: for example, DMA_PE can be fixed to the data section end, while DMA_PS let free to vary, in order to set the size of the reserved block. In this way the overhead is halved, since only one value has to be stored, at the cost of flexibility, as the reserved block would be always in a fixed position (the end of the data section, in the provided example).

A further design choice is to store only one pair of start and end addresses, outside the MAL circuit. In this way the register usage is dramatically decreased, as only one pair of registers is instantiated, at cost of system flexibility. With this choice, in fact, only a single DMA-allowed subset can be set per time. Hence, it becomes impossible, for example, to have a DMA data transfers between SMs, as it would require the setting of two memory subsets in the data sections of the involved modules. The only way to do it, is to move data from the first module to unprotected memory, an then having the second module configuring a DMA operation to read them.
Another drawback is an increased rigidity in the handling of the content of start and end addresses. Every time a software module wants to start a DMA operation, it needs to set the content of those register to be sure of allowing the DMA accesses into the intended memory subset. On the contrary, previous approaches do not require this precaution, as the ends of each memory subset were separately stored for each SM, thus never overwritten unless the proprietary SM wanted to modify its memory subset. This guaranteed that after a module set it own DMA-allowed space, it would have been available until the module disabled it. SMs were, thus,

provided with the possibility of allowing external devices to indefinitely write in their data section without any further CPU intervention. This capability is useful in those situation when data arrival is not deterministically predictable. To make the discussion more concrete, a small example is provided: imagine that a software module wants to allow data transfers from a device, with DMA capabilities, connected to a sensor. In order to do so, it first sets the boundaries of the DMA-allowed region in its data section. Then, it provides them to the device so that, when data are available, this latter can write them into the reserved space. Since the sensor can output data at any time, it is fundamental to always guarantee DMA access to that specific confined memory region inside the SM. This is possible only if each module stored its own boundaries for the DMA-allowed subsets.

Pros and cons with respect to the ones from previous solution, when:
1) The start or the end addresses of the DMA-allowed subset is fixed

Pros	**Cons**
• Register overhead is halved, since only the loose boundary has to be stored	• Reduced system flexibility in positioning the subset inside the data section

2) Only one pair of start and end addresses are allowed per time

Pros	**Cons**
• Register overhead is dramatically reduced, since only one pair of start and end addresses has to be stored	• Reduced system functionalities: – impossible to use DMA to directly transfer data between two SMs – impossible to indefinitely set a DMA-allowed memory region inside modules data sections, in order to allow background incoming of data – software overhead, as each SM needs to load the start and end addresses of its DMA-allowed memory subset before starting any DMA operation

4.6 Summary of Memory Access Rights

Memory access control rules enforced on Sancus are here summarised in Table 4.2, directly following what done in Table 2.1, from section 2.2 in referring to a generic PMA. Depending on the current value of the program counter (first column of the Table 4.2), different memory rights are granted on the text or data sections, or unprotected memory. Notice that accesses carried from others SMs are considered as same as originated from unprotected memory.
A similar table is provided also for the DMA accesses (Table 4.3). Every row in this table shows the memory permissions for each of the different branches of the MAL circuitry, as presented in section 4.5. Notice that only read and write permissions are shown, as DMA is not related with code execution. Except from the second row, memory access rights solely depends on the memory location currently pointed by the DMA address. In case no memory access control is enforced (first row), no rules are enforced and Direct Memory Access is provided to every location of the system memory. When MAL circuity is enforced, access to protected memory is fully denied (third row), or allowed only for a specific subset defined by the SM (fourth row). On the contrary, when PC-based memory access control is enforced, DMA memory permissions depends on whether the PC is inside the text section of a module, or not.

4.7 Open Problems

Section 4.5 extensively dealt with possible security solutions to prevent malicious exploitation of DMA capabilities. However, by considering that DMA extends I/O capabilities of the system, it is reasonable to assume that it increases flaws, too.

Rowhammer The possibility of exploiting an ad-hoc channel to repeatedly access specific memory locations without any CPU intervention widens the threats of Rowhammer attacks. Since the root cause of the vulnerability to these attacks is due to the increase in density and decrease in sizes of DRAM cells [20], main countermeasures to this problem are not directly related with DMA, but imply hardware or software fix as described in [23, 20].

Side Channel Attacks The world of side channel attacks is various and flourishing: from measuring the power consumption to time execution duration, these kind of attacks can break down security properties on modern processors. Spectre [25], Meltdown [28] and Foreshadow [9] are just some of the most recent attacks that showed their disruptive capabilities on high-end processors. Sancus architecture suffers from side channel attacks, too. In the Nemesis paper [10], researchers

	Memory access rights				
CPU	from \ to	Protected			Unprotected
		Entry point	Text	Data	
	Entry point	r x	r x	r w	r w x
	Text section	r x	r x	r w	r w x
	Unprotected \ Other SMs	x			r w x

Table 4.2: Memory access rights in program counter-based memory access control on Sancus. Access rights are shown for CPU memory accesses

DMA with:	Memory access rights			
No memory access control enforced		Protected		Unprotected
		Text	Data	
	DMA address poinitng to	r w	r w	r w
PC-based memory access control enforced	PC in \ DMA to	Protected		Unprotected
		Text	Data	
	Text section	r -	r w	r w
	Unprotected \ Other SM	- -	- -	r w
No access to protected memory		Protected		Unprotected
		Text	Data	
	DMA address poinitng to	- -	- -	r w
Access to specific memory locations		Protected		Unprotected
		Text	DMA_PS – DMA_PE	
	DMA address pointing to	- -	r w	r w

Table 4.3: Memory access rights in program-counter based memory access control. Access rights are also shown for DMA accesses, for all the proposed solutions from section 4.5. Among these, only the second one makes use of the PC to validate DMA accesses (second row of the table).

leveraged flaws at microarchitectural level to leak secrets from a software module. The main consideration is that each instruction take a specific number of CPU cycles to execute. Therefore a conditional path, with different instruction on the two branches, takes different time to execute, depending on which of the two branches is taken. In brief, by carefully timing the execution time, an attacker can understand which branch was taken, thus retrieve the content of the secret test condition. More specifically, this is achieved by firing an IRQ right after the conditional jump took place. Since IRQ s on Sancus are served only on completion of the current instruction, the IRQ latency depends on which instruction was executing, thus on which branch was taken.

By adding Direct Memory Access to the architecture, the side channel attack base is extended. Considering that the memory interface is shared between the DMA controller and the CPU, and that the DMA controller only operates in transparent mode[7], DMA requests are not served until the CPU stops accessing the same resource the DMA wants to access - may it be the program or the data memory. An attacker could exploit this to understand when memory is being accessed, by measuring the time required to perform a specific DMA operation. Notice that there is no difference in accessing a specific memory location or a random one in terms of delay in starting the operation, as the memory interface is shared. Thus, there is no chance to disclose the address of a specific memory location by only using this side channel vulnerability.

[7] A summary of operation modes for DMA arbiters can be found in section 2.4 *"Direct Memory Access (DMA)"*

Chapter 5

DMA Interface Implementation

The content of this chapter goes through the implementation steps required to provide the Sancus architecture with the features described in the Design and Discussion chapter. First, an explanation of the DMA interface integration on Sancus is provided; then the focus is moved on securing DMA accesses on Sancus (section 5.1).

5.1 Secured DMA Interface for Sancus on OpenMSP430

Sancus architecture is originally implemented on the openMSP430, a synthesizable 16bit microcontroller core written in Verilog. The core is compatible with Texas Instruments' MSP430 microcontroller family and can execute the code generated by any MSP430 toolchain [18].
At the moment of writing, the openMSP430 core is released at its version r211, fully integrating a DMA interface. However Sancus architecture was developed in 2013, thus it relies on an older version of the core which does not include any DMA interface. In order to extend DMA capabilities to the architecture in use, two choices are possible: to properly transfer Sancus on the current openMSP430, or to modify the already existing architecture to provide it with DMA interface. While it may seem to worsen future maintainability of the project, the selected approach is to extend the DMA interface on the already existent Sancus, rather than porting Sancus on the most recent core release. In this way portability is facilitated and the design process speeds up; furthermore, the choice is justified by the lack of relevant updates of the core after Sancus was developed. In fact, by

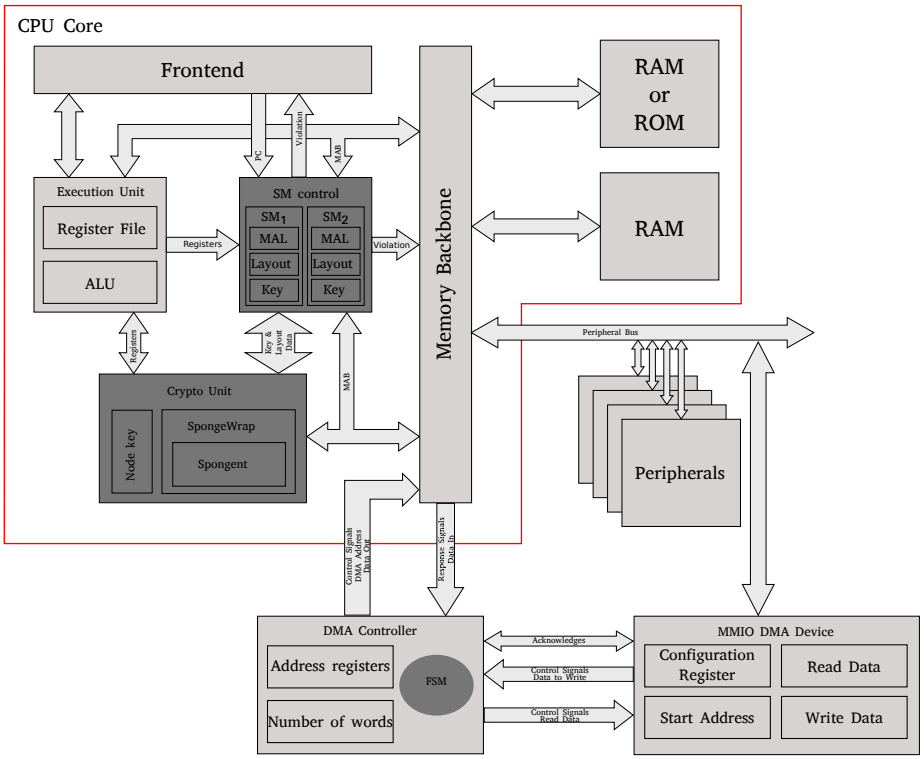

Figure 5.1: Detailed overview of the system architecture, showing the DMA controller and a peripheral with DMA capabilities, connected to it. Constituent blocks of the core are here shown. Everything related to the DMA or peripherals is outside the TCB, whose boundaries extends to the CPU, memory backbone and the system memories. In fact, no assumption is needed on DMA controller nor devices trustability.

looking into the core's changelog[1], it is possible to notice that the updates dated later than 2013 concern improvements for the compiler and simulation toolchain, fixes of the core's code for ASIC implementation, the addition of the DMA support and the extension of the number of supported interrupt requests. Considering that the latter is not a relevant concern for the developing of Sancus, it appears clear that the only major hardware change is the extension of the DMA interface support on the already existing architecture. In order to do so main modifications need to be carried out on fundamental blocks, like the core frontend, execution unit and memory backbone.

[1]openMSP430 core's changelog is available at `https://opencores.org/websvn/filedetails?repname=openmsp430&path=%2Fopenmsp430%2Ftrunk%2FChangeLog_core.txt`

5.1.1 Memory Backbone Modification

In openMSP430 architecture the memory is seen as a single block, composed by program memory, data memory and peripheral space, sequentially arranged. Each physical memory location is a 16 bits word, whereas logical memory words are 8 bits long, as explained in section 2.4.2 *"Memory in the openMSP430 architecture"*. Every time a logical memory location is accessed, physical address has to be obtained first. The memory backbone handles the translation logical addresses into physical ones, and it takes care of driving the control signals for the DRAM blocks implementing data and program memory. It is clear, then, that the memory backbone plays a major role for in extending the DMA interface on the relying architecture.

First modification consists in renaming part of the signals in the core Verilog code in order to keep continuity with the latest release of the openMSP430, for future maintainability. Then, the DMA interface is moved into the memory backbone: DMA and debug interface signals are multiplexed in the external memory signals: *ext_enable*, *ext_write*, *ext_address* and *ext_dout* are used for the usual translation of logical addresses into physical ones. Notice that the memory backbone hierarchy always considers external memory accesses with the lowest priority w.r.t. frontend and execution units. The overhead due to the extension of DMA interface in the memory backbone only consists in the multiplexing of two signals (*ext_enable* and *ext_write*) ans two buses (*ext_address* and *ext_data_in*), together with the addition of *dma_data_out* bus for read data as well as of the DMA response signals. In this sense, the choice of extending the DMA interface on Sancus, rather than to move Sancus on the newest openMSP430, is more portable. A final difference is to be noticed: in the original openMSP430 core the execution unit cannot write into the program memory, which can be modified by the external; however, in Sancus, the execution unit needs to be able to write in program memory to correctly deploy a SMs. As final remark, the memory backbone receives in input all the control signals of the DMA interface described in subsubsection 2.4.2.1 *"DMA Interface - Signals"*, and outputs the transfer complete and transfer response signals, together with the output data.
Finally, in order to not affect Sancus availability, the DMA operation must be always carried out in transparent mode[7]. This can be assured by setting the DMA priority to the LOW value; considering that this signal is driven by the DMA controller connected to the core, which lies outside the TCB (Figure 5.1), it cannot be trusted. The only way to assure the transparent mode for all the DMA operations, without further extending the TCB, is to hardwire the priority signal to LOW directly in the memory backbone.

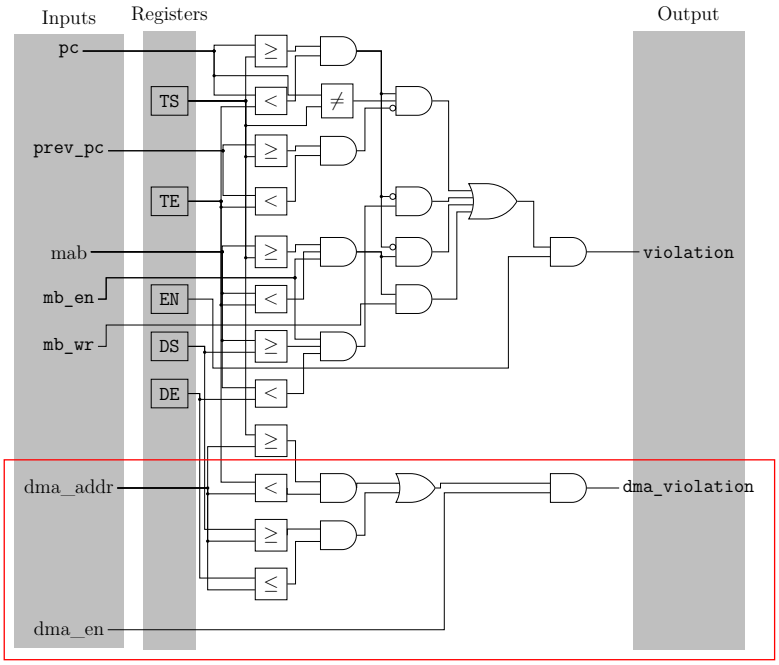

Figure 5.2: Single instantiation of the Memory Access Logic (MAL) circuit, used to enforce the memory access rules in the nodes. In the upper part of the image PC-based memory access control is enforced on the MAB. The highlighted box shows the hardware that realises the protection mechanism of subsection 4.5.3: when the DMA is enabled and its address is included between the text section start and end addresses (TS and TE), or between the data section start and end addresses (DS and DE), a violation is raised.

5.1.2 Frontend Modification

The frontend is that part of the control unit in charge of fetching and decoding the instruction; through its state machines it orchestrates the fetch-decode-execute cycle, providing the decoded instructions to the execution unit; it also handles interrupts. The implementation of the DMA interface caused one single relevant change in this block: the ability of generating an IRQ on DMA violations. The IRQ has the second highest priority of the system, after the software module interrupt request, in order to promptly intervene on unauthorized memory accesses. In doing so, the frontend is provided with the DMA violation signal, as new input from the memory backbone.

5.1.3 Execution Unit Modification

In combination with the frontend, the execution unit constitutes the CU. This fundamental block includes the interrupt logic, the register file, decoding hardware for instructions execution, the ALU as well as with the memory interface. Furthermore, it hosts the crypto unit and the Sancus' modules control. Inside this latter the memory access logic (MAL) is instantiated. The MAL circuit - already discussed in subsubsection 2.3.4.1, and whose image is here reported in Figure 5.2 for reader's ease - validates memory accesses by comparing its input with the start and end addresses of text (TS and TE) and data (DS and DE) sections of a SM, when the protection *EN* bit is asserted. A copy of the circuit is instantiated for each SM in the system, and a violation is raised in case of illegal accesses. By ORing together the violations signals coming from all the MAL circuits it is possible to check that no access to protected memory is performed.
Every time a SM is added or updated, its start and end addresses of text and data sections need to be stored in the MAL circuit; when this happens, the new values are taken from the the register file, as shown in Table 5.1, which is part of the execution unit, too.

TS	$\leftarrow$	*r12*
TE	$\leftarrow$	*r13*
DS	$\leftarrow$	*r14*
DE	$\leftarrow$	*r15*

Table 5.1

From the analysis of section 4.5 *"Protection of System Memory from DMA Attacks"*, some solutions are proposed in order to prevent the malicious exploitation of the DMA interface. The solution adopted in Sancus is to prevent DMA any access to the memory (the solution is described in subsection 4.5.3). This implies that the address of the memory location to be accessed by DMA must undergo the same MAL circuitry that validates the execution unit MAB. The challenge in doing so is to keep a low utilization of resources, as Sancus components are sufficiently register-consuming. The adopted solution aims to reuse the already instantiated MAL, extending it so that it could validate the memory address when a DMA operation is executing; in this way no additional register is instantiated.

Chapter 6

DMA Controller Implementation

The content of this chapter focuses on the implementation of the DMA controller: the component is first introduced (section 6.1), together with the pros and cons of adding it to the system. Then, the description of its internal blocks is provided (section 6.3), as well as with testbenches that show its functionalities and operation modes (section 6.6).
Finally, two devices related with the controller are analysed: (1) the controller driver, whose main task is to interface the controller with the higher level software executing on the core (section 6.4); (2) a device with DMA capabilities, that simulates a directly connected peripheral for in the simulation of a real-case scenario (section 6.5).

6.1 Overview of the DMA Controller

Substantial work of the thesis implies the design of a DMA controller to direct DMA operations. Its role is to act as an arbiter between the CPU and the external peripheral that requests memory access.
The main advantage of using a DMA controller, instead of directly connecting a peripheral to the DMA interface, is in multiplexing different devices. In order to allow this, an arbitration circuitry (Figure 6.1) collects the multiple peripherals DMA requests and resolves which one is to be served first, basing on a positional priority level, in which devices connected to the highest line are served first. Its output is stored in a register managed by the controller FSM, which is reset only on completion of the current operation. The encoded signal drives a multiplexer that connects the DMA acknowledge with the correct device which won the arbitration.

Further benefit of including the DMA controller in the system is that it incorporates all the complexity of the DMA protocol in use by the core. In this way, memory accesses can be provided even to those devices that lacks of a complex

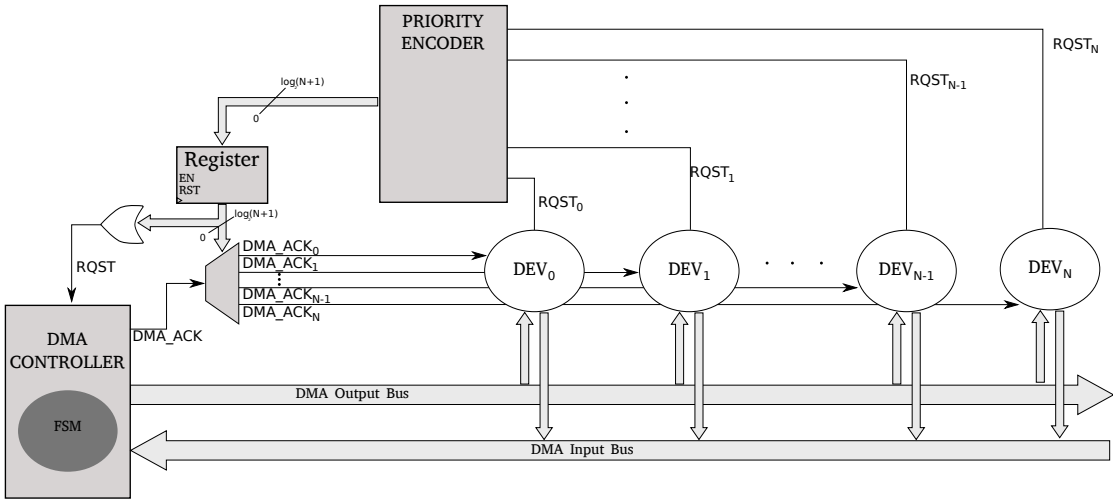

Figure 6.1: Overview of the arbitration circuitry that solves multiple DMA requests from the peripherals. The priority encoder decides which device is to be served fist, by following a positional priority level in which devices connected to the highest line are served first. Its output is stored in a register, managed by the the DMA controller FSM. The encoded signal drives a multiplexer that connects the DMA acknowledge with the correct device which won the arbitration.

logic, such as a FSM, thus incapable of driving the interface on their own.
With this approach, a DMA peripheral just needs to program the controller with the type of operation to be performed (read or write), the number of words and the starting address of the memory location to be accessed. Then, it rises a DMA request signal and waits until its request has been taken into account, and completed, by the controller.
The priority encoder collects the DMA request lines from all the connected peripherals, and resolves which device is to be served first. Then, it propagates the request to the controller, which flags its correct reception by raising the DMA acknowledge signal. The arbitration circuitry (Figure 6.1) takes care of routing this latter to the correct device, which gained access to the bus. At this point, the controller starts serving the required operation until its completion (section 6.3).

The controller can interface with MMIO peripherals as well as with devices directly connected to it. After a memory mapped operation has been set by software (section 6.4), data transfers between the controller and MMIO peripherals are carried out solely through direct read and write operations, not burdening the CPU. On the other hand, communication between the controller and a device directly connected to it happens through a simple protocol based on two acknowledge signals, which are employed in a 2-phase handshake between the two parties.

With such a system model, DMA capabilities are extended to Sancus, while supporting a wide range of devices, from simple to more complex ones.

6.2 Mode of Operation of the DMA Controller

The DMA controller is the bridge between peripherals and the system memory: it permits to move data from one domain to the other with a first-in first-out approach. The start of the operation is always initiated by a device with DMA capabilities, which also has to provide the data necessary to perform a memory access, such as the starting location and the number of words to be handled. Then the device switches in slave mode and the control passes to the DMA controller, which coordinates the required operation until its end. Because of an internal data buffer, the DMA controller manages to mask any wait state between the two domains. In fact, both the device and the core perceive the controller as a resource always available to receive data, whereas it's controller responsibility to move the received data from one domain to another by following the right communication protocol, whether it's interfacing with the core or a device. The controller is designed to always operate in transparent mode,[7] to not reduce Sancus availability. If an high priority DMA operation is started, it triggers an halt request which stops the CPU from fetching further instructions or receiving any interrupt. An attacker could exploit this feature to endanger the architecture availability, by repeatedly starting DMA operations to monopolise the resource. This eventuality is not acceptable for critical applications, as the one adopted for industry or automotive, where the usage of a non-responsive architecture could cause serious damages; thus the choice to not implement burst or cycle stealing modes in the DMA controller for Sancus.
Notice that the interface between the controller and the core flags any attempt of accessing outside of memory mapped areas through the response signal. If this is the case, the DMA controller immediately stops the current operation, flags the error to the device and resets its internal registers so that any data can't leak or influence future operations.
A more detailed description of the DMA controller functionalities and implementation is found in section 6.3.

6.3 Implementation of the DMA Controller

The analysis from the previous section introduces the DMA controller in a top-down approach, starting from the required functionalities, to the architecture that realises them. Thus, a sketch of the DMA controller emerged, presenting the

design challenges. In this section the focus is more on the implementation of those features and the strategies used in doing so. The internals of the DMA controller are here presented, organized in different subsections, each for a different component of the controller. The approach is bottom-up, with each component separately designed and tested by ah-hoc testbenches, whose purpose is to solely imitate the openMSP430. A full system test is carried out after the design of all the fundamental blocks has been completed, and can be found in section 6.7 *"Attack Scenario on DMA-Secure Sancus Implementation"*. Here, the controller is connected to the DMA interface of the real openMSP430 core, and it handles the operations required by a single device, whose implementation is discussed in section 6.5.

From the discussion of chapter 4, DMA controller's main properties emerged; those can be summarised as follows:

1. it must be able to be minimally programmed by a requesting device to perform single or multiple read or write operations;

2. in case of multiple DMA accesses, the controller must transparently transfer data between the openMSP430 and the requesting device;

3. the handling of the DMA interface must totally rely on the controller, unburdening both the requesting device and the CPU from doing so. This includes the need for the controller to operate in transparent mode, thus to support wait states flagged from the openMSP430 (see section 2.4 for more detail on DMA operation modes).

A more detailed overview of the system is provided in Figure 5.1. In the image, the constituent blocks of the Sancus architecture are shown, together with the TCB boundaries.

Before proceeding with a description of the controller's internals, it's advised to briefly review how read and write operations are carried out in the DMA protocol in use between the openMSP430 core and the DMA controller, already introduced in subsubsection 2.4.2.2 *"DMA Interface - Protocol"*.

6.3.1 DMA Protocol - Read Operation

During a read operation the *DMA_ADDR* and the control signals need to remain stable until the end of the transfer is flagged by an *HIGH* value on *DMA_READY* signal; this holds even in case of wait states inserted by the openMSP430. In Figure 6.2 correct multiple read operations are shown: considering that the designed controller is a synchronous machine, a transfer complete is only *'sensed'* on the

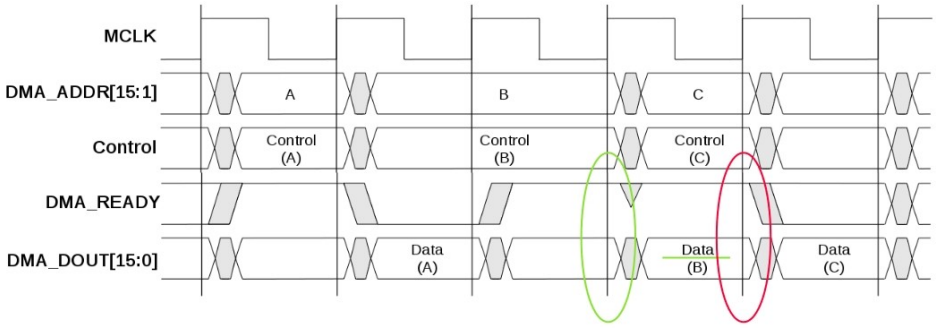

Figure 6.2: Timing diagram of a read operation. The image underlines the timing constraint to be respected when interfacing with the DMA protocol in use on the openMSP430 (detail of the protocol can be found in subsubsection 2.4.2.2). During a read operation, data are outputted on the clock cycle after the *DMA_READY* was asserted (first ellipse in the image). Thus, the controller can only sample the *DMA_DOUT* bus on the successive clock cycle (second ellipse in the image).

next clock cycle after the *DMA_READY* was raised *HIGH*; until that moment, the address and the control signals cannot change, as the read operation is not finished yet.

It's important to underline that data are outputted on the next clock cycle after the transfer complete signal was asserted; this means that the controller can sample the *DMA_DOUT* bus only during the second clock cycle after the transfer completed. This is clearly shown in Figure 6.2: by focusing, for example, on datum B it is possible to see that the DMA data out bus is driven with the data on the first front edge of the clock after the *DMA_READY* signal was raised high, i.e. after the wait state ended (first ellipse in Figure 6.2). Thus, a device connected to the DMA interface can only sample the data on the successive front edge of the clock, i.e. two clock cycle after the transfer complete signal was risen (second ellipse in the Figure 6.2).

6.3.2 DMA Protocol - Write Operation

As for read operations from previous section, the end of a write transfer is flagged by a rise of the *DMA_READY* signal; until that moment, the *DMA_ADDR*, the control signals and the output data bus need to remain stable, and this holds even in case of wait states. In Figure 6.3 correct multiple write operations are shown: again, a transfer complete is sensed only on the next clock cycle after the *DMA_READY* was raised *HIGH*.

An asymmetry between read and write operations can be noticed. From the

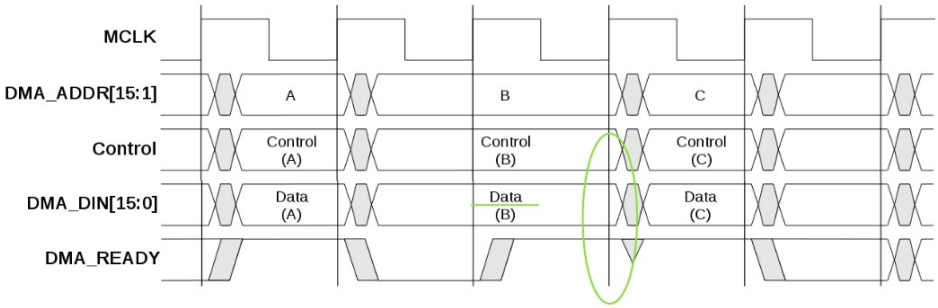

Figure 6.3: Timing diagram of a write operation. The image underlines the timing constraint to be respected when interfacing with the DMA protocol in use on the openMSP430 (detail of the protocol can be found in subsubsection 2.4.2.2). During a write operation, data are stored on the clock cycle after the *DMA_READY* was asserted (green ellipse in the image). Thus, from the controller prospective, a write operation lasts only one clock cycle, if no wait states are inserted.

controller's point of view, a read operation, with no wait state, ends two clock cycle after the *DMA_READY* was asserted. On the contrary, during a write operation, the assertion of the *DMA_READY* signals that data have been stored in the main system memory; thus a write operation, with no wait states, ends in only one clock cycle. It's important to underline this asymmetry as it causes a different handling of the read and write branches of the ASM charts of Figure 6.4. Even if the number of states used for both the operations is the same (four states for each), a difference is present: in the write branch the *SEND_TO_MEM0* state is only used once as configuration state, whereas its read-branch counterpart *LOAD_DMA_ADD* is actively involved in the read transfer, as revealed by presence of an active *DMA_EN* signal. This means that a read operation indeed requires two states of the FSM to be executed.

6.3.3 DMA Controller ASM Chart

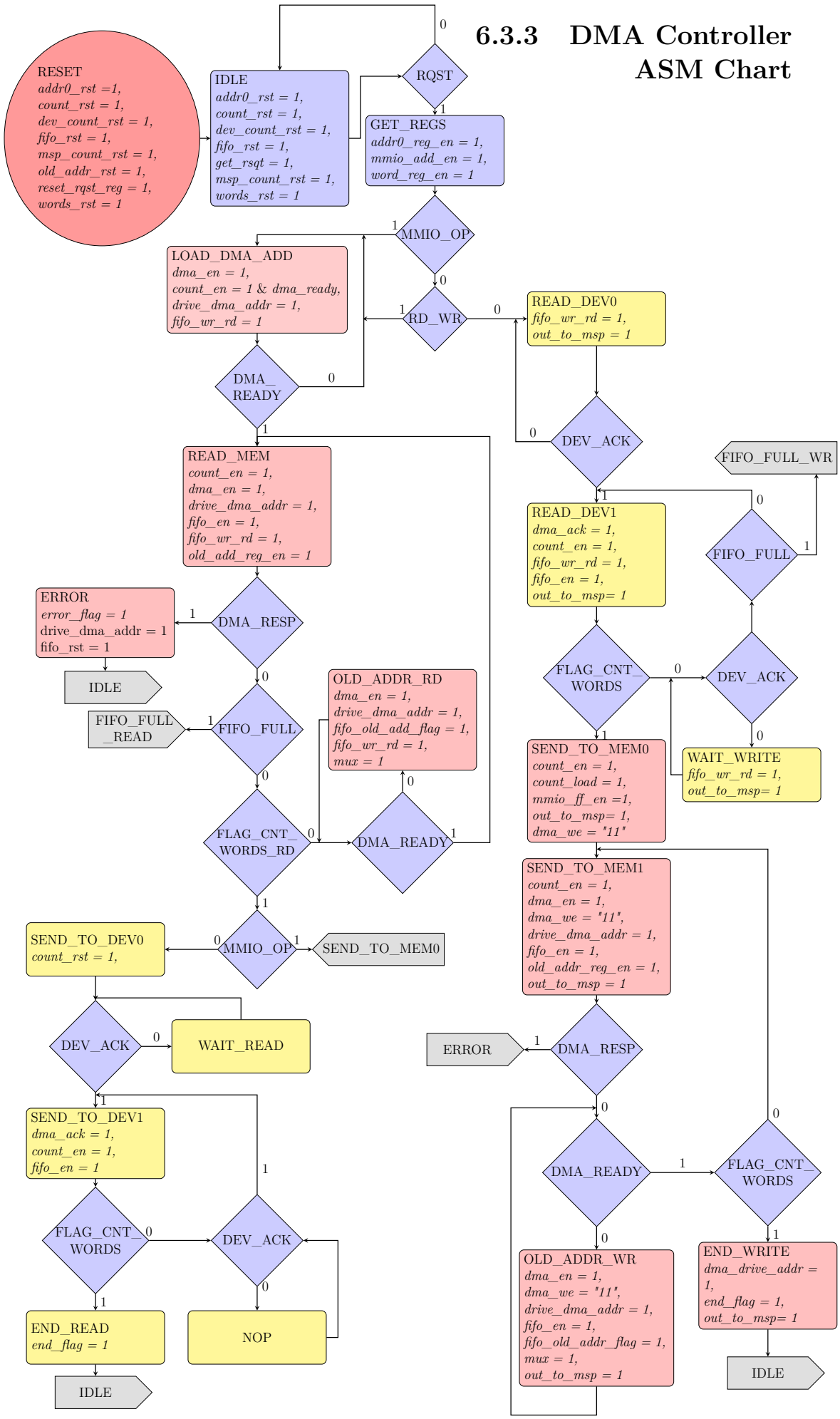

Figure 6.4: DMA controller FSM. Yellow blocks refer to the interface between the DMA controller and the openMSP430, whereas pink ones to the interface between DMA controller and device.

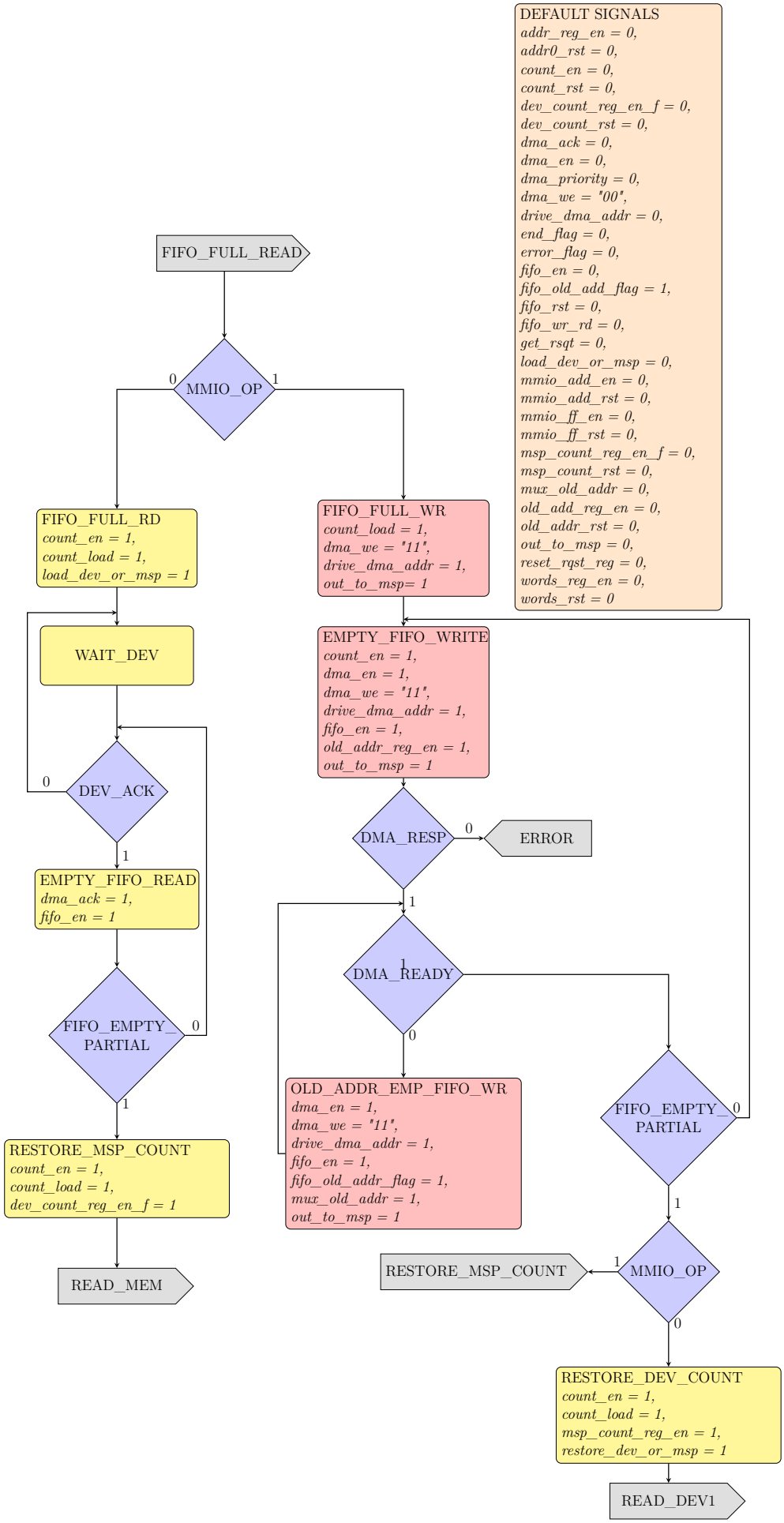

Figure 6.5: ASM chart of the DMA FSM: detail of FIFO_FULL branches, and view of default values for the signals of the FSM. When controller's internal memory gets full, the executing operation is temporarily paused and FIFO is emptied; the already stored data are sent to the receiving party, until the memory has been emptied. Then, the operation can resume to its previous state.

6.3.4 DMA Controller Data Path

6.3.4.1 Internal Registers

A first mention is for the two registers, *START_ADDRESS* and *NUM_WORDS*, shown in the upper part of the data path. These registers respectively stores the starting address for the DMA operation and the number of consecutive words in case of multiple accesses; consider that consecutive accesses are all of the same type, i.e. once a read operation has started, it cannot be switched into a write one, or vice versa. Programmability of the controller is thus achieved: a device, in fact, only needs to write into those register the desired values, before rising a DMA start request. From there on, it's controller's responsibility to complete the required operation. The value of the starting address is never modified by the controller, and it's only updated when a new operation is requested from an external device starts. Thus, in order to sequentially access memory location, a counter is synchronously incremented and the count value is added to the starting address as an offset.

In case of wait states from the core, the controller must be able to recover from where it stopped. When a wait state occurs, the transfer complete signal *DMA_READY* is asynchronously driven *LOW* from the memory backbone internal logic. However, the controller is a synchronous machine, thus it *'senses'* the wait state only on the next clock edge, at the same moment as the counter increments its value. It's only at this point that the controller enters a different state, disables the count enable, and starts waiting for the openMSP430 to be ready again to resume the transfer (find the ASM chart for the controller's FSM in Figure 6.4). However, when the wait state ends and operation resumes, the counter incremented and the accessing address is pointing to a different memory location. A shrewdness to bypassing this problem is to store a delayed version of the address in the *OLD_ADDR* register: in this way, it's always possible to know which memory location was previously being accessed. A mux, controlled by the controller's FSM, selects one of the two addresses to be outputted as the DMA address.

The internal memory of the controller is set to be far smaller than the system memory: it would make no sense, in fact, to instantiate many registers just to allow a one-shot data transfer (find internal memory implementation in subsubsection 6.3.4.2). Furthermore, considering that the target architecture is a low-end microcontroller, it results necessary to keep resource utilisation at minimum. Hence, the need to provide the DMA controller with the ability of temporarily interrupt the receiving of further incoming data when its memory gets full, so that it can empty it by starting sending the already stored data to the receiving party.

All of this is supervised by the DMA controller: specifically this functionality is implemented in the two branches of the FSM reported in Figure 6.5.
Once the memory has been emptied, the operation must resume to its previous state. In order to achieve so, two register are instantiated: *DEV_COUNT* and *MSP_COUNT*. When the data transfer between the controller and the device, or between the controller and the core is paused, the value of the counter is store into the corresponding register, before the data buffer starts getting emptied. In this way, the transfer can be resumed from where it stopped, by simply loading the stored value into the counter itself.

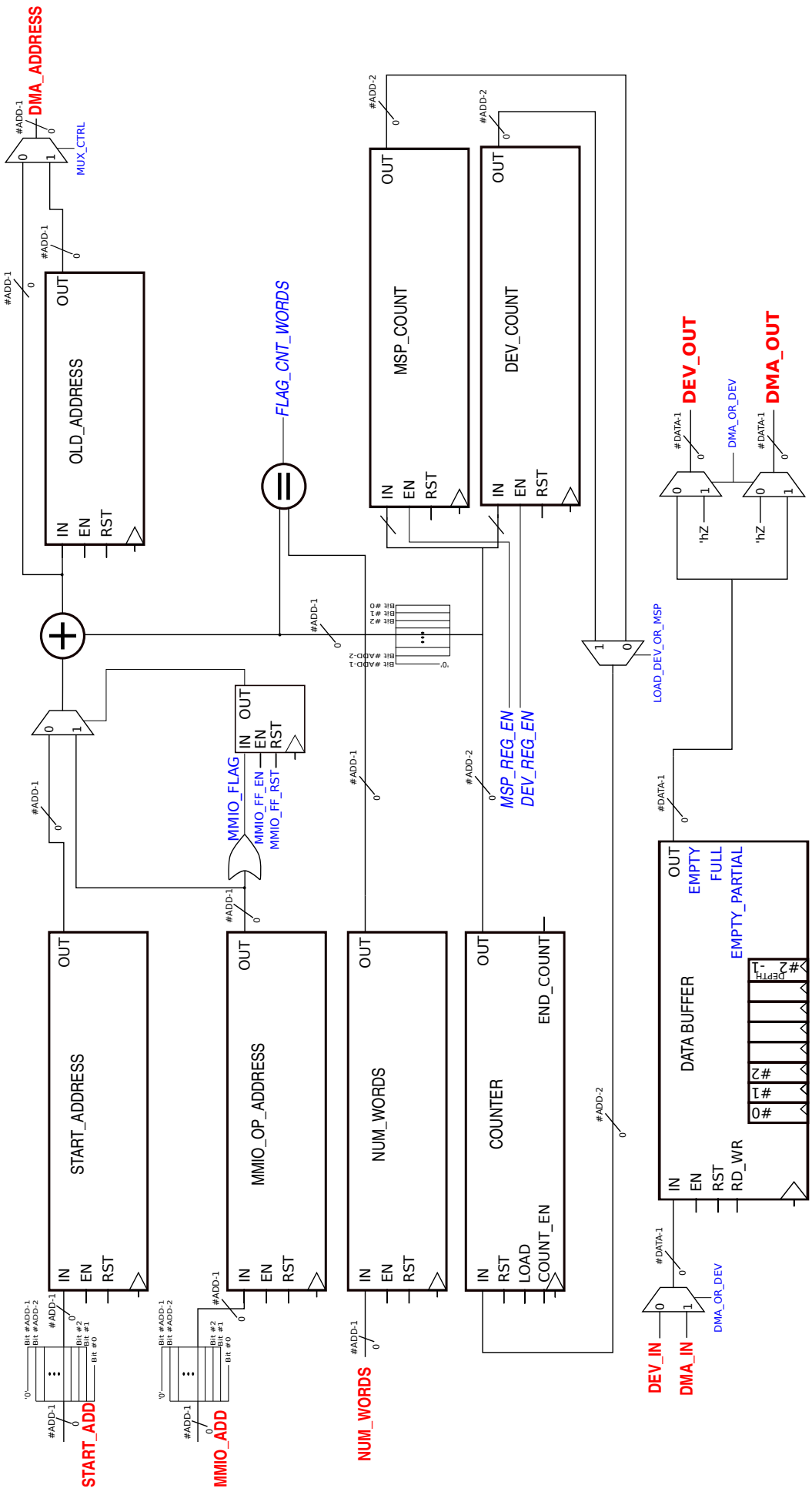

Figure 6.6: The fundamental components composing the DMA controller, together with the signals ruling its FSM behaviour are here shown. Inputs and outputs are labelled in bold, whereas all the control signals and the flags from the state machine are in italic. All the clocked devices are connected to the system clock, and all the enable and reset signals are directly driven by the controller FSM. Wires for these latter are not drawn for the sake of simplicity.

6.3.4.2 Data Buffer

The controller memory is implemented as a data buffer, whose depth is configurable during instantiation. The use of the buffer improves controller responsivity, since both the device and the core would consider it as a resource always available to receive data, until the memory fills. When this occurs, an internal flag drives the controller FSM into one of two branches of Figure 6.5, depending on whether the event happened during a read or write operation; data are sent, in a first-in first-out order, to the receiving party, either the device or the core, and the memory gets freed. Once a certain percentage of the buffer - set by the user at the instantiation - has been emptied, the controller resumes to its previous state as described at the end of subsection 6.3.4.
A first remark about the data buffer concerns the choice of its depth, as it requires an appropriate calibration basing on the expected average data exchange rate on the DMA bus. In fact, an oversized data buffer would be empty for most of the time, resulting in a waste of registers. On the other side, an undersized buffer would fill too quickly, forcing the controller to often stop collecting new data to empty its memory. The default choice is to have the *BUFF_SIZE = DMEM_SIZE / 512*, so that when a 16-kB the data memory is used, the controller internal buffer would be 32-byte deep. The empty percentage is set by default to one eighth of the buffer size.

A second remark concerns a peculiar design choice about the buffer registers, caused by the need to comply with the DMA protocol for write operations (Figure 6.3). During a wait state, the openMSP430 drives the *DMA_READY* high

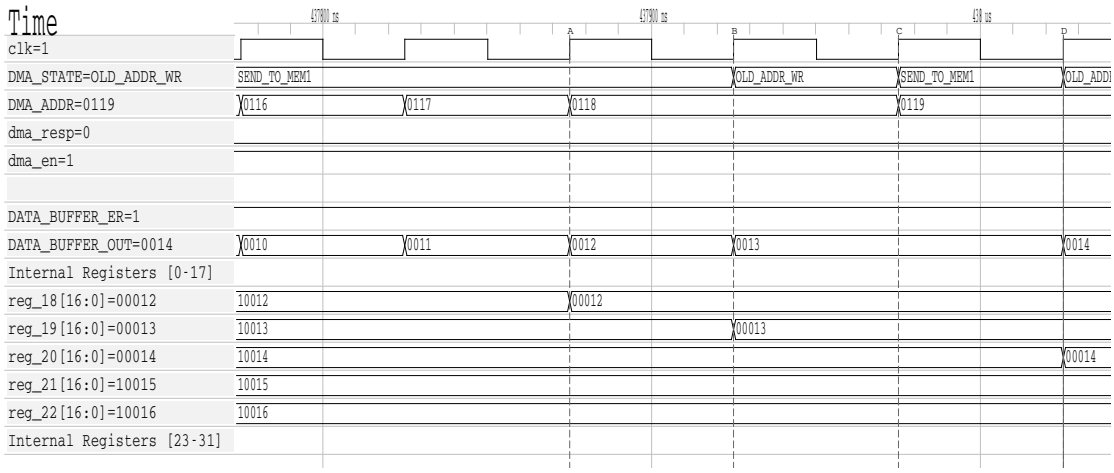

Figure 6.7: openMSP430 - DMA protocol: violation when synchronous registers are used, at markers C and D.

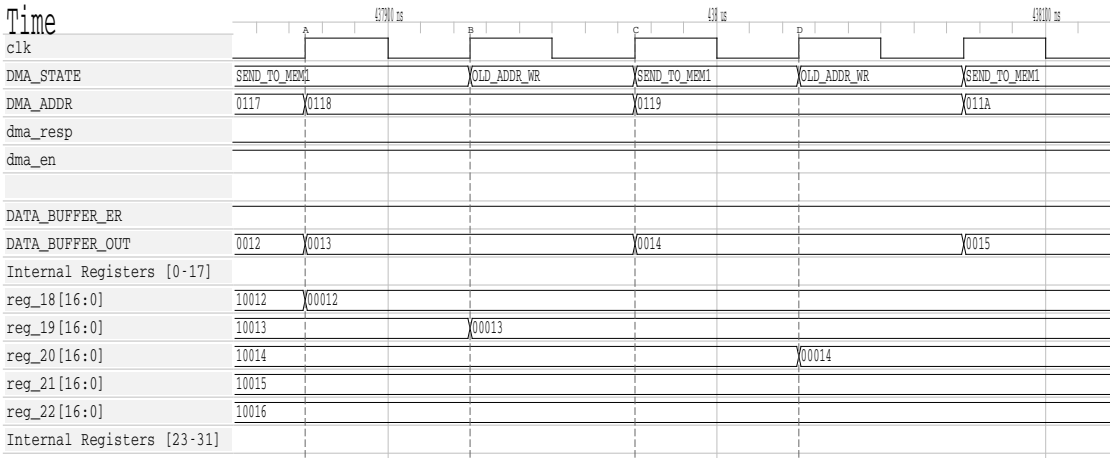

Figure 6.8: openMSP430 - DMA protocol: no violation when asynchronous registers are used.

to flag to the controller the correct sampling of the data (marker B of Figure 6.7). However, the controller is a synchronous machine: it senses the change of the ready signal only on next positive edge of the clock, and resumes from the wait state, reactivating its internal components. Simultaneously, the openMSP430 core samples the data on the *DMA_DIN* bus (marker C of Figure 6.7). The problem appears at this moment, as the sampled data is not the correct one. What happens is that the data is still the previous one (number 13), since the data buffer synchronously updates its output only on the next clock edge (marker D of Figure 6.7). The solution to this subtle timing problem is to provide the data buffer internal registers with an asynchronous output. In this way, the output correctly switches, to value 14, in correspondence of the first clock cycle after exiting the wait state (marker C of Figure 6.8).

6.4 DMA Controller Driver

In Figure 5.1 an overview of the system is provided. It is possible to notice that the DMA controller inputs are not connected to the openMSP430, but directly coming from the peripheral in use at the moment. However, in case of DMA operation involving a MMIO device, it is necessary to set the starting address, the number of words and the type of the operation (read or write) from the high level software. Therefore, the controller is provided with a driver, which handles the configuration of the controller internal registers with values from the software. Specifically, the initialization of a DMA operation implies that:

- the higher level software stores the starting address and the number of words

into the the memory-mapped registers of the driver. Then it informs the driver with the type of the operation to be carried out (read or write).

- After that, the driver is triggered and a DMA start request is performed on the controller input bus.

In this way, the controller is provided with the two stating addresses of the memory locations, where the data transfer will occur. It automatically handles all the steps required to complete the operation, without burdening the CPU.
Here are shown the two C functions used to configure the controller, through the driver. Respectively, Listing 6.1 is used when the requesting device is directly connected to the DMA controller, whereas Listing 6.2 is used for MMIO devices operations.

```
void asm_config_op( uint16_t num_of_words, uint16_t address,
    uint16_t op_code)
{

        asm(" ; Define memory addresses  \n\t"
                ".equ START_ADDR_REG , 0x0100 \n\t"
                ".equ N_WORDS_REG    , 0x0102 \n\t"
                ".equ CONFIG_REG     , 0x0104 \n\t"

                " ; Start operation           \n\t"
                " mov %0             , &START_ADDR_REG \n\t"
                " mov %1             , &N_WORDS_REG    \n\t"
                " mov %2             , &CONFIG_REG     \n\t"
                :  //no outputs
                : "m"(address), //inputs
                  "m"(num_of_words),
                  "m"(op_code));
}
```

Listing 6.1: Code snippet showing the C function used to configure the DMA controller, through the driver, in case of operations requested by a directly connected device.

```
void asm_config_mmio_op( uint16_t num_of_words, uint16_t
    stat_addr, uint16_t mmio_addr, uint16_t op_code)
{
  asm(" ; Define memory addresses  \n\t"
        ".equ START_ADDR_REG , 0x0100 \n\t"
        ".equ N_WORDS_REG    , 0x0102 \n\t"
        ".equ CONFIG_REG     , 0x0104 \n\t"
        ".equ MMIO_START_ADD , 0x010A \n\t"

        " ; Start operation           \n\t"
        " mov %0             , &START_ADDR_REG \n\t"
        " mov %3             , &MMIO_START_ADD \n\t"
        " mov %1             , &N_WORDS_REG    \n\t"
        " mov %2             , &CONFIG_REG     \n\t"

        :  //no outputs
        : "m"(address), //inputs
          "m"(num_of_words),
          "m"(op_code),
          "m"(mmio_addr));
}
```

Listing 6.2: Code snippet showing the C function used to configure the DMA controller, through the driver, in case of memory mapped operations.

6.5 Device with DMA Capabilities

Due to the need of verifying DMA controller functionalities and of simulating a real-world scenario for the testbenches, a simple device has been designed to perform memory accesses on software requests.
It is a memory-mapped I/O device controlled by higher level software executing on the machine, thought as an extension of the driver from section 6.4.
It consists of few registers devoted to memory access requests; a configuration register which stores the device state so that it can be accessed by the higher level software controlling the device; two one-bit acknowledge signals, one as input coming from the DMA controller, and the other as output from the device to the controller.

The *start_address* register contains the memory location where the memory access starts; *num_words* register stores the number of words to be read/written. Both these registers are sent to the DMA controller at the start of the operation, since it's the controller that handles the whole memory transfer, once device request to access memory has been acquired.
Write and *read* registers store, respectively, the data to be written into- and the

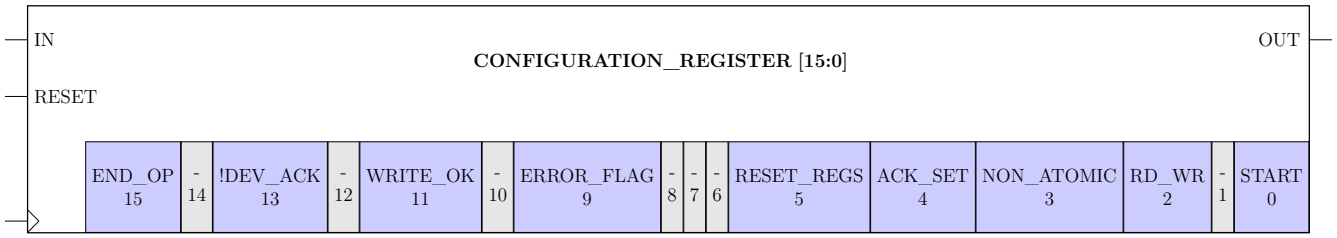

Figure 6.9: Configuration register used by the DMA device. The function of each single bit composing the register is here shown. Due to register the higher level software is capable of transparently exchange data with the DMA controller. The start of operation can be programmed by only writing into the memory mapped *STRAT_ADDRESS*, *NUM_WORDSs* and *CONFIG_REG* registers. By reading the content of the configuration register software becomes aware of the progress of the ongoing operation. Be aware that the content of the register is internally updated by the device; in this way it can keep track of the operation and correctly handle the communication with the DMA controller through a 2-phase handshake protocol (section 6.1), without direct intervention of the software layer in driving the signals.

data read from- memory. They are used to provide a communication channel between the device and the higher level program.

The configuration register (Figure 6.9) is accessed both by the software and the device itself, and stores the state of the device. When the higher level software wants to start an operation, it programs this register with the proper opcode. The device itself will update the content of the register according to its internal state; this is entirely done by the hardware, so that it is transparent to the software layer. Finally, by reading back the register content, the software can learn information about the progresses of the current operation and, thus, interact with the device: error and end-of-operation flags, read or write select bit, as well as with detailed information on the current operation are contained in this 16 bits register.

6.5.1 Overview of DMA Read and Write Operations

Data transfers between the device and the controller follow a 2-phase handshake protocol on two acknowledge signals: device acknowledge is automatically controlled from the *ACK_SET* bit of the configuration register, without software intervention.

During a write operation into the system memory, the device plays an active role, as it has to provide the controller with the data to be written (subsubsection 6.6.2.1). Once all the data have been written into the controller buffer, device stops being

an active party, and the control is deferred to the controller, which writes the data into the system memory (subsubsection 6.6.2.2).

On the contrary, once a read operation has been correctly required (which includes the sending of the start address, the number of words and the read request to the controller), the device plays a passive role, as it waits for the controller to retrieve the data from openMSP430 memory (subsubsection 6.6.1.1). Only once data have been collected, the device interfaces itself with the controllers to receive them (subsubsection 6.6.1.2): on data arrival, a change in the configuration register informs the higher level software controlling the device. Data are, thus, saved in the read register, and it is software care to save them in the new assigned memory location.

6.6 DMA Tesbenches

In this section results of the testbenches, carried out for both read and write operations, are shown. For each of the two type of operation, a general view of the main signals involved is given. Before the start of an operation, the controller is always provided with the number of words to be handled and the starting address of the memory locations to be accessed. Then, depending on the type of the request, it moves along one of the two branches of Figure 6.4.
Finally, notice that all the DMA accesses are considered to only address unprotected memory locations. A test showing memory access rules violation is shown in section 6.7.

6.6.1 DMA Controller Read Branch

A general view of the operation is shown in Figure 6.10, from its start until the end. It corresponds to the left branch of the device FSM shown in Figure 6.4 and it consists of:

1. The DMA controller interfaces with the openMSP430 DMA interface to read the requested data from the system memory (subsubsection 6.6.1.1).

2. The DMA controller interfaces with the device which started the operation and sends the data to it (subsubsection 6.6.1.2).

6.6.1.1 Read from System Memory

When reading from the system memory, the controller follows the DMA interface protocol in use on the openMSP430 (Figure 6.2): first, it requests read access to

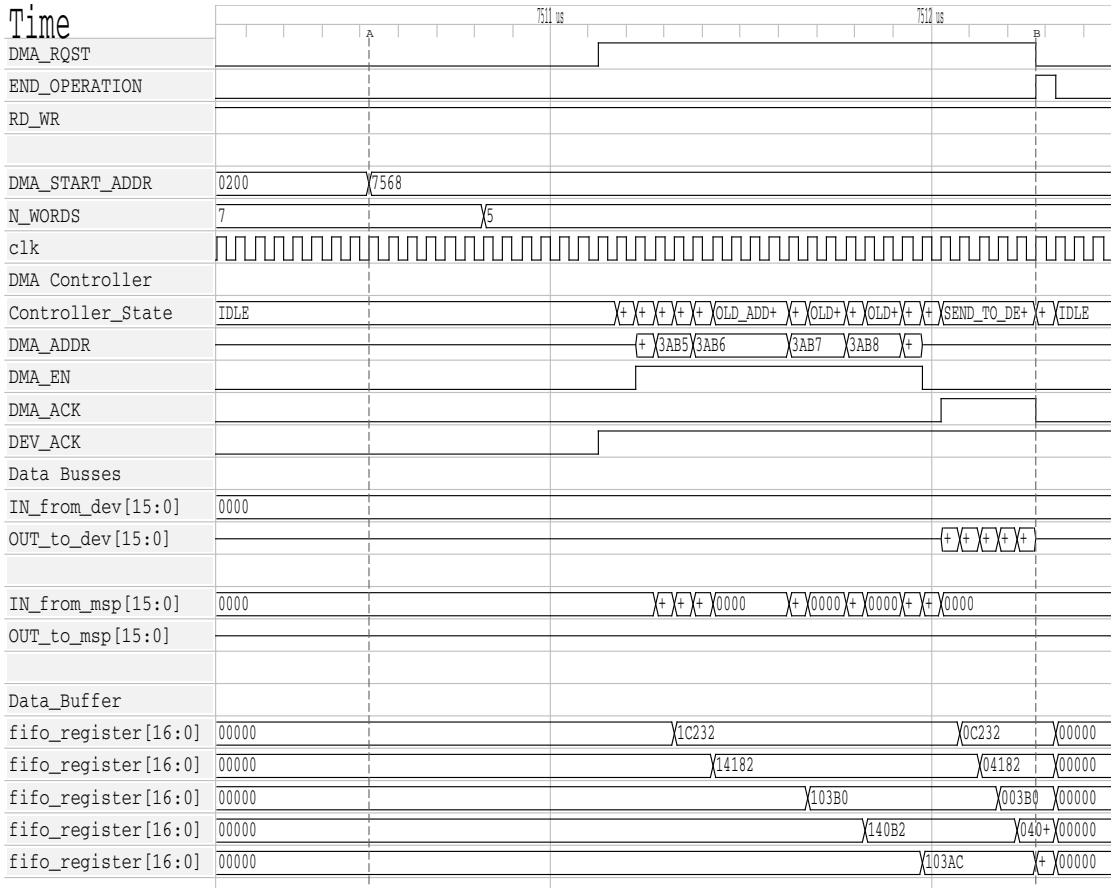

Figure 6.10: General overview of a read operation. After having set the start address and the number of words to be read, the request signal is raised (marker A) and the operation is started. Its end is flagged by the controller, at marker B.

a memory location by driving the *DMA_ADDR* and setting the *DMA_WE="00"* (at marker A of Figure 6.11). If the addressed location is not part of protected memory, no violation is raised in the the MAL of Figure 5.2. If the memory is available at moment, the *DMA_READY* signal is kept high and no wait state is inserted. Hence, the memory backbone maps the logical address from the controller into a physical one, and drives the openMSP430 output bus with the content of that memory location (marker B of Figure 6.11).
Finally, the values is sampled and stored into the controller data buffer on the next clock cycle (marker C of Figure 6.11). Notice that a read operation, with no wait states, takes two clock cycles to be executed. This delay is necessarily inserted as the openMSP430 requires time to sample the DMA address and to drive the correct data on the output bus. Therefore, the controller can only store the incoming data on the successive clock edge. However, while waiting for the data, the controller

can already drive a further read request (marker B of Figure 6.11) in order to speed up in case of multiple read operations. The new request, in fact, would be sensed by the openMSP430 interface on the same clock edge when the controller stores the previous data (marker C of Figure 6.11).
When a wait state is entered, the DMA address and the control signals driving the DMA interface are kept stable at their current value. In this way, when the system memory become available, it can provide the requested data. The openMSP430 flags it by rising the *DMA_READY* signal, and the controller samples the data on the next front edge (marker D of Figure 6.11). In the eventuality the controller data buffer completely fills, the operation is temporarily paused to start emptying the controller internal memory, as described in subsection 6.6.3.
A final remark about the data buffer concerns the way data are stored: a *'1'* is written in the first bit of the register that saves an incoming data. In this way the data buffer can keep trace of the filling state of its memory and, when full, flag it to the controller

6.6.1.2 Write to a DMA Device

The controller follows a 2-phase handshake when interfacing with devices directly connected to itself. Figure 6.12 shows the data transfer from the controller to the device. The operation only starts when the device acknowledge signal is raised high, meaning that the device is ready to store data. At this point, the controller becomes active and drives the output bus with the data to be sent; in doing so, it asserts its acknowledge signal to inform the device that a data has been sent (marker A of Figure 6.12). As soon as the device acknowledge signal is driven low, the controller enters in wait state (marker B of Figure 6.12). Notice that the data buffer is emptied on the clock cycle after the data has been sent to the device(marker B).
The controller keeps waiting until the *DEV_ACK* is raised again (marker C of Figure 6.12). Notice that the output data bus is in high impedance for all the wait, since more than the current device is connected to it (Figure 6.1). Therefore it makes sense to have the controller driving the output only when requested from the receiving device. This latter is informed of data arrival by the controller acknowledge signal which, unlike the bus, is routed exclusively to the active receiving party. In this way the data, which are meant to be sent to the active device, remains for a single clock cycle on the bus. Although this cannot be considered a 'safe' mechanism, it's a rudimental attempt of confidentiality.

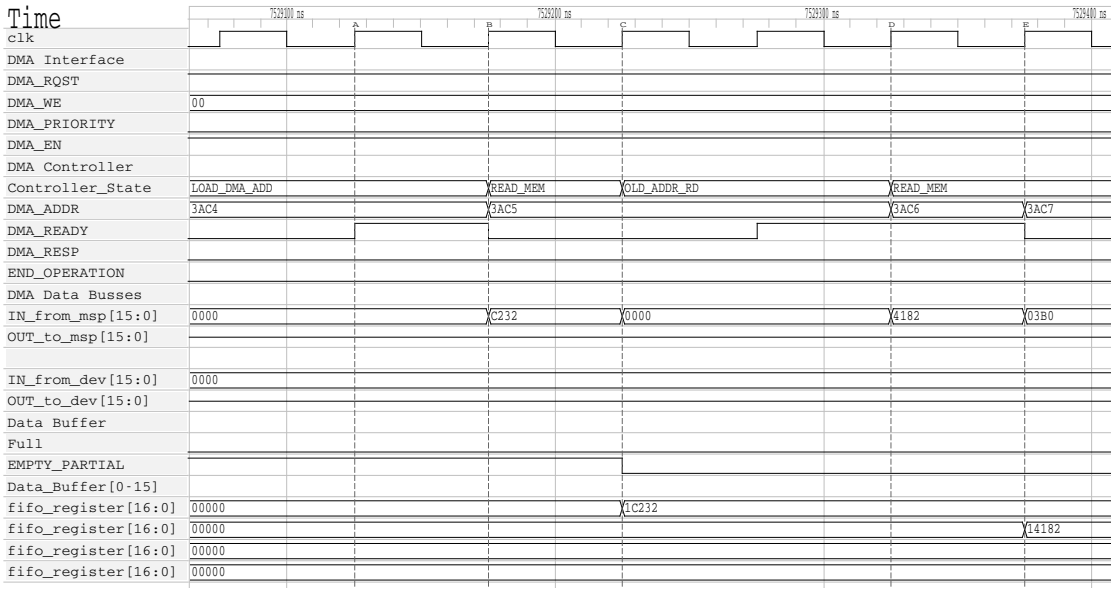

Figure 6.11: Reading data from the openMSP430 memory. The controller requests the data (marker A). If the DMA address is not pointing to a protected memory location, the openMSP430 outputs the requested data on its bus (marker B). Finally, data are stored into the controller data buffer (marker C).

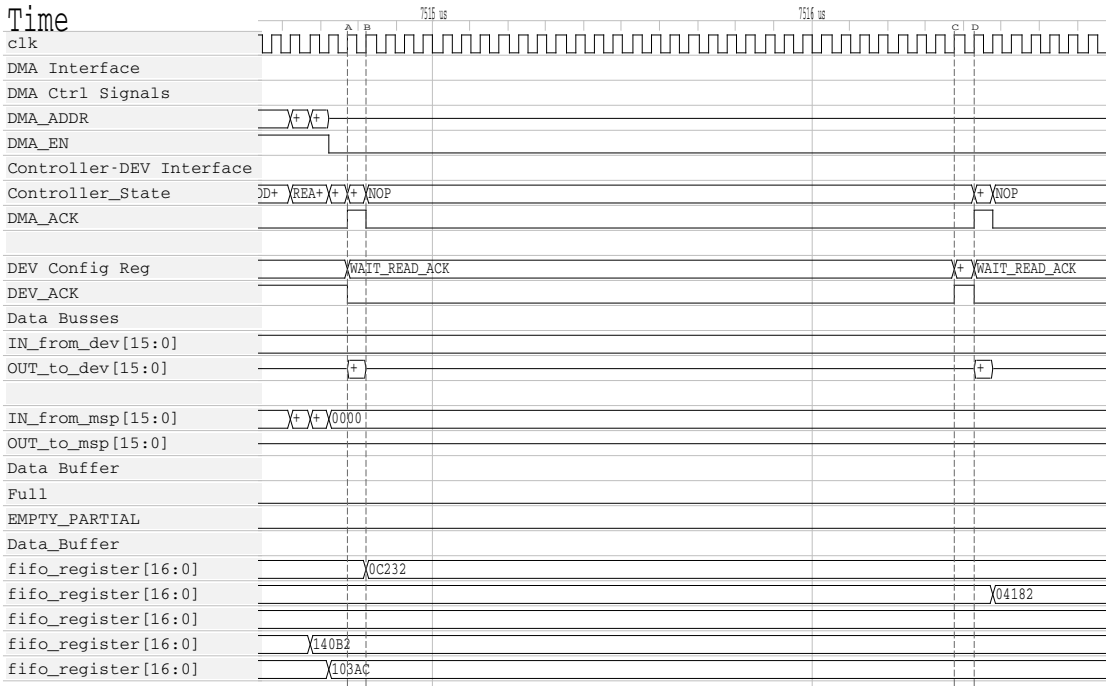

Figure 6.12: Sending data to a device directly connected to the controller. The communication protocol between the two implies a 2-phase handshake: the device requests data by raising its acknowledge signal (marker A); the controller drives the output data and flags the sending through its own acknowledge signal (marker B).

Figure 6.13: General overview of a write operation. After having set the start address and the number of words to be read, the request signal is raised (marker A) and the operation is started. Its end is flagged by the controller, at marker B.

6.6.2 DMA Controller Write Branch

A general view of the operation is shown in Figure 6.13, from its start until the end. It corresponds to the right branch of the device FSM shown in Figure 6.4 and it consists of:

1. The DMA controller interfaces with the device which started the operation and retrieves the data to be written into the system memory (subsubsection 6.6.2.1).

2. The DMA controller interfaces with the openMSP430 DMA interface to write into the system memory (subsubsection 6.6.2.2).

6.6.2.1 Read from a DMA Device

The controller follows a 2-phase handshake when interfacing with devices directly connected to itself. Figure 6.14 shows the data transfer from the controller to the device. The operation starts only start when the device acknowledges the availability of new data (marker A of Figure 6.14), which are driven on the controller

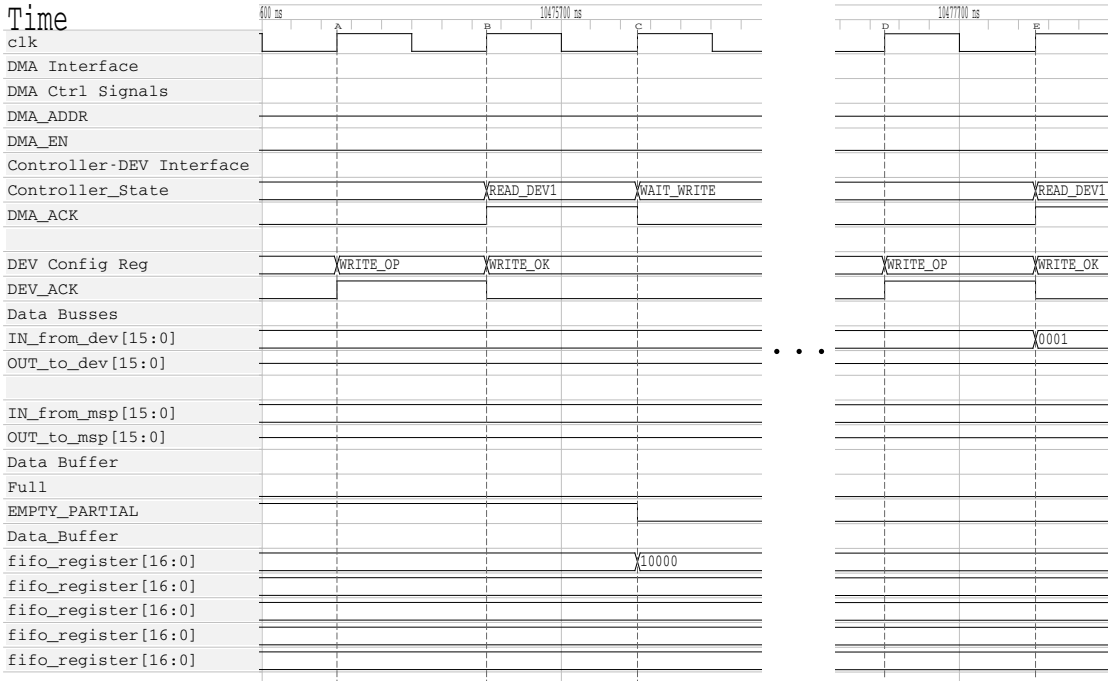

Figure 6.14: Receiving data from a device directly connected to the controller. The communication protocol between the two implies a 2-phase handshake: the device flags the availability of new data by raising its acknowledge signal (marker A); the controller stores them flags correctness of the operation through its own acknowledge signal (marker B).

Figure 6.15: Writing data to the openMSP430 memory. The controller drives the data output bus and requests a write operation (marker A). If the DMA address is not pointing to protected memory, the openMSP430 saves the incoming data (marker C). When *DMA_READY* is driven low, a wait state is inserted (marker B). Markers C and D show two successive writings, with no wait state inserted.

input data bus. At this point, the controller becomes active, activates the data buffer and rises its acknowledge signal to flag the correctness of the operation (marker B of Figure 6.14). Notice that data are actually stored into the controller internal memory only on the successive clock cycle, as expected from registers with synchronous input (marker C of Figure 6.14). In the provided example, the device is implemented to keep driving the data on the input bus until the next operation; this choice depends on how the device is internally designed, and does not concerns nor affect the controller.

6.6.2.2 Write into System Memory

When writing into the system memory, the controller follows the DMA interface protocol in use on the openMSP430 (Figure 6.3): first, it requests the access to a memory location by driving the *DMA_ADDR*, together with the data bus to the memory (at marker A of Figure 6.15). If the addressed location is not part of protected memory, no violation is raised in the the MAL of Figure 5.2. Hence, the memory backbone decodes the logical address received from the controller into a physical one and stores the received data into at the desired location (at marker C of Figure 6.15). In case the ready signal is driven low (at marker B of Figure 6.15), a wait state is inserted. When this happens, the controller keeps driving the address and the data bus until the core samples them and flags it by raising the *DMA_READY* high again (at marker C of Figure 6.15).
Notice that, when a data is read, the first bit of the corresponding register is set to *'0'*. In this way the data buffer can keep track of the emptying state of its memory. This capability is fundamental to correctly flag the partial or full emptying of the memory.
Finally, when no wait state are inserted, wait operations are handled one per clock cycle. At marker C a write operation to the specified address, is requested; on the next clock edge the ready signal is high, meanign that data has been correctly sampled (at marker D of Figure 6.15); therefore, the successive write request is carried on by driving the address of the new destination to be written.

6.6.3 Emptying the Controller Data Buffer

Figure 6.16 and Figure 6.17 show the details of the controller internal behaviours when emptying the data buffer, corresponding to the two branches of the controller FSM, as shown in Figure 6.5.
Respectively, Figure 6.16 shows the timing diagram of emptying operation in which data are outputted to external device. On the contrary, in Figure 6.17 data are removed from the internal data buffer and stored into the system memory.

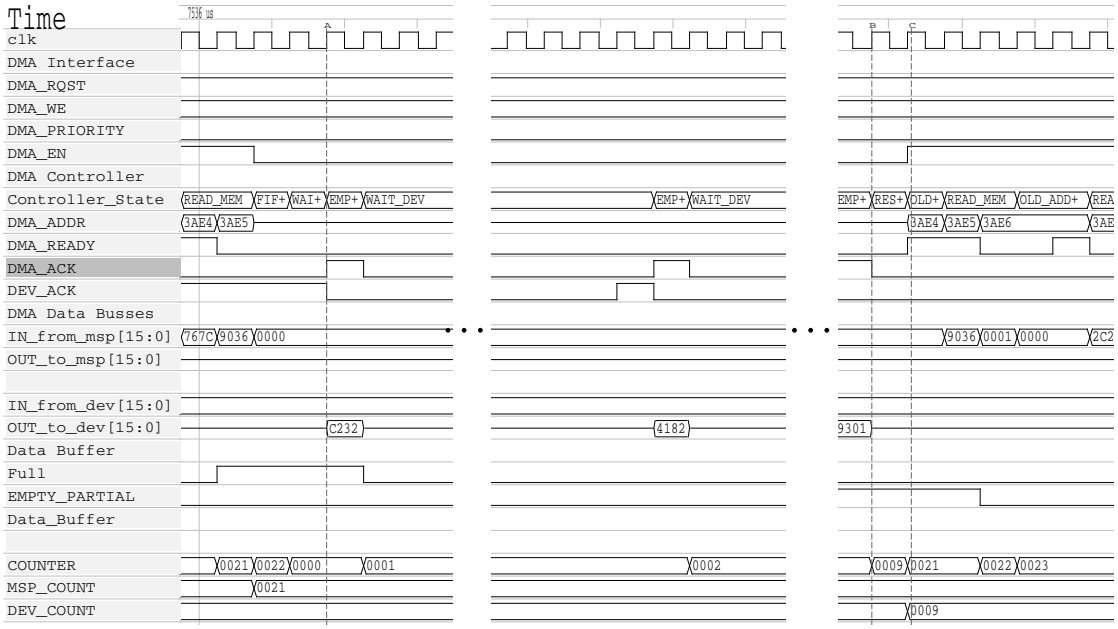

Figure 6.16: Emptying the controller data buffer by outputting data to an external device.

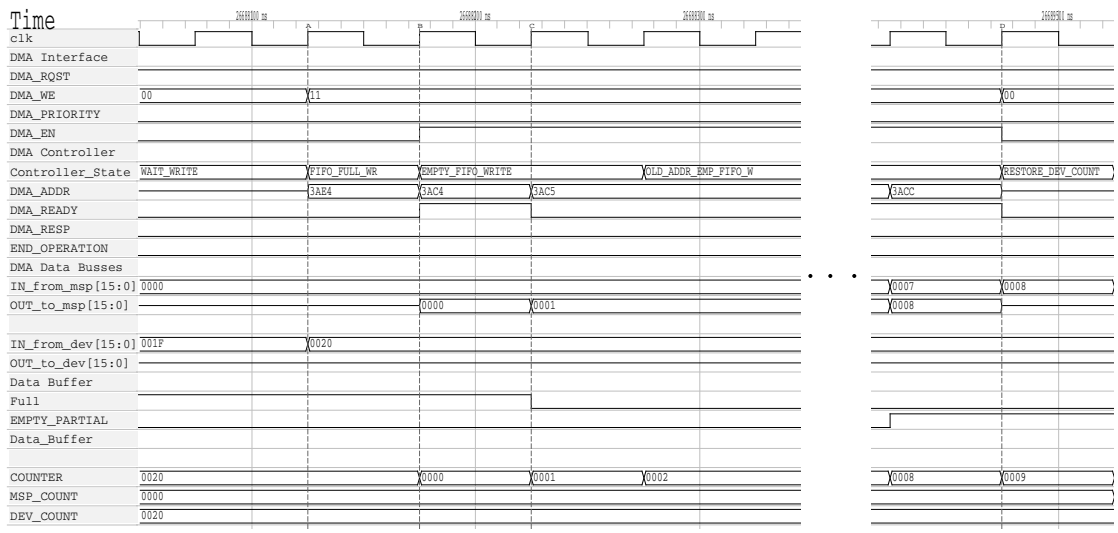

Figure 6.17: Emptying the controller data buffer by outputting data to system memory.

6.6.3.1 Emptying the Buffer - Output to External Device

On the first clock cycle after the data buffer full signals has been asserted, the controller temporarily stops the current operation to empty the internal memory and enters the left branch FSM emptying procedure, as shown in Figure 6.5. Here, the data transfer is handled as a normal write operation to an external device, by following the 2-phase handshake between the device and the controller: this latter proceeds by outputting the data only when required from the device (marker A of Figure 6.16). This information is carried by the device acknowledge signal: when it's value is high it means that the controller can send data; when driven low, wait states are inserted and the controller waits until the next dare are requested by the device.
When the the empty flag, the controller exits the emptying procedure. In doing so, the value of the *DEV_COUNT* register is loaded with the number of data written during the operation. In this way, next accesses to the device memory will start from that value (9, in this example). Simultaneously, the value of MSP_COUNT is loaded, and the previous operation resumes.

6.6.3.2 Emptying the Buffer - Output to System Memory

At the first available clock front edge on which the data buffer full flag is sensed, the controller temporarily stops the current operation to empty the internal memory and enters the right branch of FSM emptying procedure, as shown in Figure 6.5. Here, the data transfer is handled as a normal write operation to the system memory, by following the DMA interface protocol in use on the openMSP430 core (subsubsection 6.6.2.2). The controller first drives the data output bus and the DMA address (at marker A Figure 6.17). If no memory violation is raised, the core would sample the data on the next clock cycle (at marker B in Figure 6.17). Wait states are handled as usual: when the *DMA_READY* is driven low (at marker C Figure 6.17), the controller starts waiting until the memory becomes available again; then the operation continues.
Finally, on the rising of the empty flag, the controller exits the emptying procedure. In doing so, the value of the *MSP_COUNT* register is loaded with the number of data written during the operation. In this way, next accesses to the system memory will start from that value (9, in this example). Simultaneously, the value of *DEV_COUNT* is loaded, and the previous operation resumes.

6.7 Attack Scenario on DMA-Secure Sancus Implementation

When Sancus is provided with a secure DMA support, for example by implementing the solution from subsection 4.5.3 (*"Exclude DMA from Protected Memory"*), every illegal access to protected memory is detected and prevented. A DMA violation signal is rised from the MAL circuitry and the core DMA bus is automatically zero-driven, to prevent any leak of data. In Listing 6.3 a possible attack scenario is depicted: it proceeds by configuring a DMA read from a SM text section and, then, injects malicious code into it. The results of the attack, displayed on a terminal, are shown for both the Sancus versions with or without the secure DMA implementation. Specifically, the left column of Listing 6.4 shows the outcomes of the attack, when carried on a secure DMA architecture. The right column depicts the results of same attack when carried on Sancus with a direct DMA implementation (the same of section 3.2). Finally, Figure 6.18 highlights what happens from an architectural point of view: when the DMA tries to access a protected memory

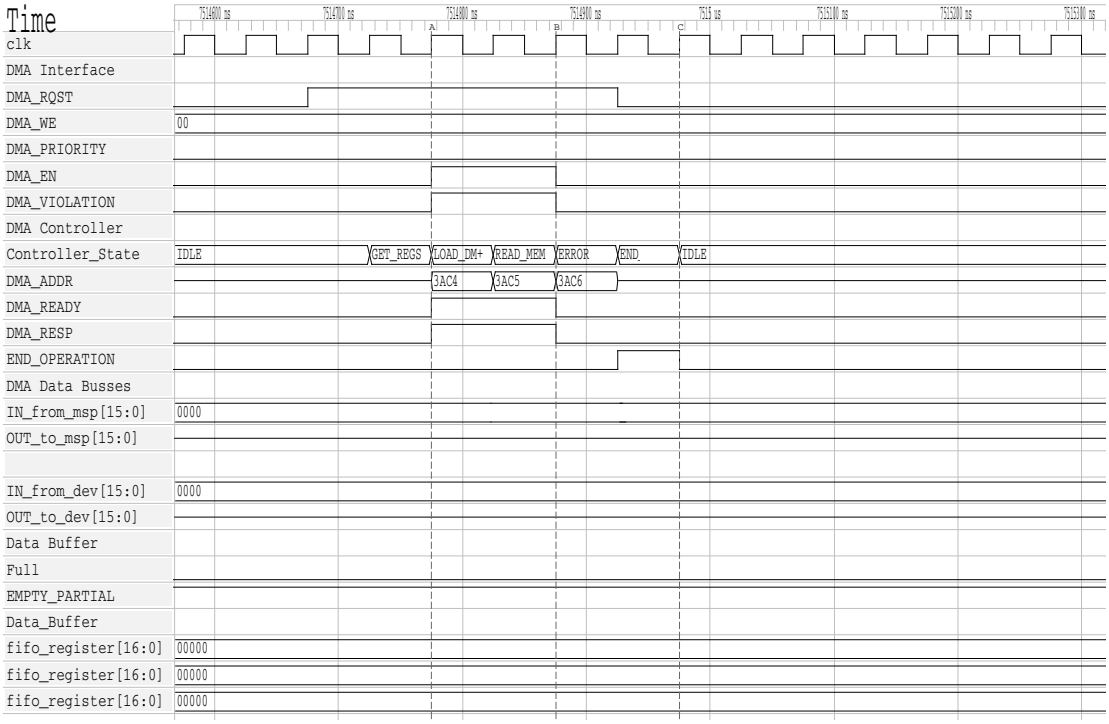

Figure 6.18: Detail of the DMA violation signal when an illegal access to protected memory occurs. On DMA illegal accesses to the memory, the core DMA output bus is driven to zero, and the DMA controller exits the current operation, reporting an the error.

location, the MAL asserts a DMA violation signal (marker A of Figure 6.18). The DMA output bus of the openMSP430 core is automatically driven to zero. Furthermore, the controller does not sample any data, and proceeds by exiting the operation going through the *ERROR* state. Notice that it's possible to write an ISR to be started on DMA violations, if necessary. To this end, it is sufficient to follow the same procedure explained in Listing 3.5, from subsection 3.2.1.

```
//================================
//       INITIALIZATION
//================================
msp430_io_init();

// ALLOCATE MEMORY TO STORE THE
   DATA
Save_data_location = (uint16_t *)
   malloc(N_DATA*sizeof(uint16_t)
   );
if (save_data_location == NULL) {
    printf("[main.c] impossible to
        allocate enough memory
       for saving the data!\n");
    EXIT();
}
//================================
//        ENABLE THE SMs
//================================
pr_info("enabling sensor/reader
   SMs..");
sancus_enable(&sensor);
pr_sm_info(&sensor);
sancus_enable(&reader);
pr_sm_info(&reader);

      .
      .
      .

// GET SM IDENTITY
get_struct_val(&reader, &ts, &te,
   &ds, &de, &id, &vendor_id,
   name);
printf("SM ID: %d \n TS: 0x%.4x -
   TE: 0x%.4x \n DS: 0x%.4x - DE
   0x%.4x \n", id, ts, te, ds, de
   );
```

```
//================================
//        START THE ATTACK
//================================

pr_info("starting dma illegal
   access...");
pr_info1("start reading into SM%d'
   s text section...\n",id);
dma_read(ts, N_DATA,
   save_data_location);

// PRINT THE LEAKED SECRET DATA
for (i = 0; i< N_DATA; i++)
    printf("Data%d at addr. 0x%.4x
        \t 0x%.4x \n",i, print_add
       , *(data_saved+i) );

// WRITE INTO SM TEXT SECTION
pr_info1("[main.c] start writing
   into SM%d's text section...\n",
   id);
dma_write(ts, N_DATA, data_to_send
   );

printf("[main.c] start reading
   into SM%d's text section after
   having written...\n",id);
dma_read(ts, N_DATA, data_saved);

for (i = 0; i< N_DATA; i++)
    printf("Data%d at addr. 0x%.4x
        \t 0x%.4x \n",i, print_add
       , *(data_saved+i) );
```

Listing 6.3: Code snippet of *'main.c'* functions. On the left, the initialization procedure of the attacks it's shown. Variables are here allocated and the SMs protection is enabled. On the right, the code that implements the attack is shown. It first proceeds by reading multiple data from the SM text section and, then, it injects malicious code in it.

```
//================================
//       ENABLE THE SMs
//================================
[main.c] enabling sensor/reader SMs
.................................

SM key: d3d41bc0a099864a
SM sensor with ID 1 enabled     : 0
   x78c4 0x7902 0x0190 0x0198
.................................

SM key: 0d7b5bc961d7696a
SM reader with ID 2 enabled     : 0
   x7588 0x78c2 0x02aa 0x03b4
// GET SM IDENTITY
SM ID: 2
 TS: 0x7588 - TE: 0x78c2
 DS: 0x02aa - DE 0x03b4

//================================
//       START THE ATTACK
//================================
[main.c] DMA illegal access...
[attacker] Reading into SM2 text
   section...
[attacker] Num. of Words: 7
Data0 at addr. 0x7588: 0xc232
Data1 at addr. 0x758a: 0x4182
Data2 at addr. 0x758c: 0x03b0
Data3 at addr. 0x758e: 0x40b2
Data4 at addr. 0x7590: 0x03ac
Data5 at addr. 0x7592: 0x03b2
Data6 at addr. 0x7594: 0x4211
[attacker] Writing into SM2 text
   section...
[attacker] Num. of Words: 7
...

[attacker] Reading into SM2 text
   section after having written...
[attacker] Num. of Words: 7
Data0 at addr. 0x7588: 0x0000
Data1 at addr. 0x758a: 0x0001
Data2 at addr. 0x758c: 0x0002
Data3 at addr. 0x758e: 0x0003
Data4 at addr. 0x7590: 0x0004
Data5 at addr. 0x7592: 0x0005
Data6 at addr. 0x7594: 0x0006
// ============================
//      SIMULATION PASSED      |
// ============================
```

```
//================================
//       ENABLE THE SMs
//================================
[main.c] enabling sensor/reader SMs
.................................

SM key: d3d41bc0a099864a
SM sensor with ID 1 enabled     : 0
   x78c4 0x7902 0x0190 0x0198
.................................

SM key: 0d7b5bc961d7696a
SM reader with ID 2 enabled     : 0
   x7588 0x78c2 0x02aa 0x03b4
// GET SM IDENTITY
SM ID: 2
 TS: 0x7588 - TE: 0x78c2
 DS: 0x02aa - DE 0x03b4

//================================
//       START THE ATTACK
//================================
[main.c] DMA illegal access...
[attacker] Reading into SM2 text
   section...
[attacker] Num. of Words: 7
Data0 at addr. 0x7588: 0x0000
Data1 at addr. 0x758a: 0x0000
Data2 at addr. 0x758c: 0x0000
Data3 at addr. 0x758e: 0x0000
Data4 at addr. 0x7590: 0x0000
Data5 at addr. 0x7592: 0x0000
Data6 at addr. 0x7594: 0x0000
[attacker] Writing into SM2 text
   section...
[attacker] Num. of Words: 7
...

[attacker] Reading into SM2 text
   section after having written...
[attacker] Num. of Words: 7
Data0 at addr. 0x7588: 0x0000
Data1 at addr. 0x758a: 0x0000
Data2 at addr. 0x758c: 0x0000
Data3 at addr. 0x758e: 0x0000
Data4 at addr. 0x7590: 0x0000
Data5 at addr. 0x7592: 0x0000
Data6 at addr. 0x7594: 0x0000

// ============================
//      SIMULATION PASSED      |
// ============================
```

Listing 6.4: The output of the terminal in which the attack was launched is here reported. On the left, the results are shown for a naive implementation of the DMA with full access to system memory. Hence, the attacker manges to fully leak text section secrets and, then, to inject external code into it. Finally, it proceeds to read through the text section again, to make sure the data were correctly injected. On the contrary, the right column shows the results for a secure DMA implementation. Every DMA access is, thus, prevented from the MAL circuitry. The read data are all zeros, both before and after writing. This means that every DMA access has been successfully prevented.

Conclusions

The increased connectivity of computing devices, and the even greater percentage of lightweight embedded devices networked in the IoT, fostered researched to look for solutions to prevent malicious exploitation of those. Protected Module Architectures (PMAs) arose as a promising line of research that prevents any illegal access to the memory regions labelled as private. Their combination with the PC based memory access control mechanism extended the offered protection to low-end devices. However, a limitation of these kind of architectures is the absence of Direct Memory Access (DMA) support. This latter unburdens the CPU from handling accesses to the memory for those peripherals supporting it. To this extent, DMA is useful to improve system performance and responsiveness when operating peripherals that perform input or output operations on big data objects, such as storage controllers, graphics cards or cameras.

The work of this thesis focuses on the analysis and discussion of solutions to include DMA support in Sancus, a lightweight security architecture for networked devices. The proposed approaches range from not including DMA in the system at all, through granting DMA accesses to unprotected memory only, or to relax some security guarantees and allow a confined DMA access to modules data sections.
A brief summary of the explored ideas, which are fully discussed and presented in chapter 4, is here provided:

1. Enforce Memory Access Logic (MAL) control by validating DMA accesses on the current value of the Program Counter (PC). However this way of proceeding revealed to be flawed, since it permits memory rights escalation attacks. By definition, the CPU is not aware of DMA accesses. Therefore, the value of the PC is free to vary during DMA operations, which is the reason why DMA is considered an improvement in the first place. However, in the framework of PMAs, this allows situation in which DMA grants access to protected memory just because PC entered a SM, fully breaking isolation and confidentiality guarantees offered by Sancus. Although this approach is excluded from possible solutions, it is still meaningful to underline the need to validate DMA accesses independently from the PC.

2. By following the approach pursued in well known high-end architectures, such as Intel SGX or Iso-X, a suitable solution is to totally exclude DMA from protected memory. In this way, modules isolation and confidentiality are restored, as no attacker can exploit the DMA bus to tamper with the software memory without rising a violation.

3. Although the solution from above allows to successfully extend DMA capabilities on Sancus architecture, without directly affecting its security properties, it might be too restrictive. SMs could benefit from using DMA to exchange data without burdening the CPU. At the same time, this must not invalidate the security guarantees of the intended modules. Thus, a possible approach is to allow SMs to voluntarily relax isolation and confidentiality properties for a confined memory subset inside the data sections, where peripherals with DMA capabilities - both memory mapped and external ones - are allowed to access.

Even if the proposed solutions differ from each other, a common ground appeared, i.e. the inclusion of DMA interface on PMAs comes with the risk of providing full access to the system memory, without any CPU control. Thus, the need to enforce a memory protection mechanism on the DMA bus urgently emerged, to prevent the system exposure to severe vulnerabilities which would eventually invalidate SMs isolation and confidentiality.

Specifically, the last proposed solution from subsection 4.5.4 broadened the possibilities to new ways of including the DMA on PMAs, differing from what proposed by the counterparts from both the embedded and the high-end world, where the trend respectively is to not provide PMAs with any DMA support, or to entirely prevent DMA accesses to protected memory.

A final important remark concerns the Trusted Computing Base (TCB) considered in each design step. A general goal of the computer security is to keep the TCB as small as possible. This same approach has been followed in this work, by trusting only on fundamental components for the fulfilment of Sancus, and leaving outside of the TCB everything concerning the DMA interface, including the peripherals with DMA capabilities as well as the DMA controller.

Future Work

Future work would primarily focus on the implementation of the solution from subsection 4.5.4, and on further developments that could possibly originate from it. To this extent, for example, it would be interesting to investigate the possibility of allowing SMs to selectively grant access to DMA peripherals to their own "DMA-allowed" subsets. Most likely, this is unfeasible with the current system model,

considering that selectivity on peripherals can be achieved only if the arbiter that rules on them is trusted. In this case, in fact, it could receive and store private informations, like the identity or the ID, of the module that started - or allowed - a specific DMA operation; then, it could run it with the same context, i.e. with the same memory access permissions, of that specific module.

In the system model discussed in this thesis, the arbiter is the DMA controller. Therefore, the consequences of including the DMA controller into the TCB would be paramount for the future growth of the architecture, and definitely worthy to analyse. If the DMA controller can be trusted, then a new scenario opens for the previously discarded solution, vulnerable to memory access rights escalation attacks. A trusted DMA controller, capable of storing the ID of the SM module that requested the operation, could operate memory accesses with the same access rights of the module itself. Furthermore, if peripherals are provided with IDs, the controller would be able to selectively grant DMA capabilities to specific devices.

Bibliography

[1] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors, 2012.

[2] G. A. Akpakwu, B. J. Silva, G. P. Hancke, and A. M. Abu-Mahfouz. A survey on 5g networks for the internet of things: Communication technologies and challenges. *IEEE Access*, 6:3619–3647, 2018.

[3] Godfrey Akpakwu, Bruno Silva, Gerhard P. Hancke, and Adnan Abu-Mahfouz. A survey on 5g networks for the internet of things: Communication technologies and challenges. *IEEE Access*, 5:3619 – 3647, 12 2017.

[4] Niels Avonds, Raoul Strackx, Pieter Agten, and Frank Piessens. Salus: Non-hierarchical memory access rights to enforce the principle of least privilege. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, pages 252–269, Cham, 2013. Springer International Publishing.

[5] Aaron Ballman. Attributes in c. Open PDF, June 2016. Available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2049.pdf`.

[6] Adam Boileau. Hit by a bus: Physical access attacks with firewire. Online resource, 2006. Available at `https://security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`.

[7] Rick Boivie. Secureblue++: Cpu support for secure execution. Technical report, IBM Research Report, 2012. Available at `https://domino.research.ibm.com/library/cyberdig.nsf/papers/E605BDC5439097F085257A13004D25CA/$File/rc25287.pdf`.

[8] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: Tiny trust anchor for tiny devices. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 34:1–34:6, New York, NY, USA, 2015. ACM.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX: Security Symposium (USENIX Security 18)*, page 991 –1008, Baltimore, MD, 2018. USENIX Association.

[10] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 178–195, 2018.

[11] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.*, 1(1):50–60, February 2004.

[12] Guillaume Vissian Christophe Devine. Compromission physique par le bus pci. In *Proceedings of SSTIC '09*. Thales Security Systems, June 2009.

[13] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[14] Karim El Defrawy, Daniele Perito, Gene Tsudik, and et al. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In *In: Proceedings of the 19th annual network and distributed system security symposium*, pages 5–8, 2012.

[15] Kaspersky Lab Encyclopedia. History of malicious programs. Online resource. Available at `https://encyclopedia.kaspersky.com/knowledge/history-of-malicious-programs/`.

[16] AB Ericsson. Cellular networks for massive iot: Enabling low power wide area applications. Ericsson White Paper, 2016. Available at `https://www.ericsson.com/assets/local/publications/white-papers/wp_iot.pdf`.

[17] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 190–202, Washington, DC, USA, 2014. IEEE Computer Society.

[18] Oliver Girard. openmsp430. Online resource, 2016. Available at `http://opencores.org/project/openmsp430`.

[19] LLVM Developer Group. Clang. Online resource, 2016. Available at `http://clang.llvm.org`.

[20] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 245–261, May 2018.

[21] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. Soteria: Offline Software Protection within Low-cost Embedded Devices. In *Proceedings of the 31th Annual Computer Security Applications Conference*, 2015. UnivIS-Import:2015-10-26:Pub.2015.tech.IMMD.lehrst.soteri.

[22] Intel. 8237a high performance programmable dma controller (8237a-5). Online resource, 1993. Available at `https://pdos.csail.mit.edu/6.828/2012/readings/hardware/8237A.pdf`.

[23] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 5:1–5:6, New York, NY, USA, 2017. ACM.

[24] Ramya Jayaram Masti, Claudio Marforio, and Srdjan Capkun. An architecture for concurrent execution of secure environments in clouds. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, pages 11–22, New York, NY, USA, 2013. ACM.

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[26] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.

[27] Kaspersky Lab. What is a trojan virus? Online resource. Available at `https://www.kaspersky.com/resource-center/threats/trojans`.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user

space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, 2018. USENIX Association.

[29] P. Maene, J. Gotzfried, T. Muller, R. de Clercq, F. Freiling, and I. Verbauwhede. Atlas: Application confidentiality in compromised embedded systems. *IEEE Transactions on Dependable and Secure Computing*, page 1, 2018.

[30] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67:361–374, 2018.

[31] MarketWatch. Proofpoint uncovers internet of things (iot) cyberattack. Online article, January 2014. Available at `https://www.marketwatch.com/press-release/proofpoint-uncovers-internet-of-things-iot-cyberattack-2014-01-16`.

[32] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: Tamper-proof code execution on legacy systems. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[33] Andrew Martin. The ten-page introduction to trusted computing, 2008.

[34] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[35] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[36] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[37] Christian N. Klein Michael Becher, Maximillian Dornseif. Firewire: all your memory are belong to us. Online resource, 2005. Available at `https://cansecwest.com/core05/2005-firewire-cansecwest.pdf`.

[38] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle, 2015.

[39] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Transactions on Privacy and Security (TOPS)*, 20(3):7:1–7:33, September 2017.

[40] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, April 2015.

[41] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[42] Dries Schellekens. Design and analysis of trusted computing platforms. Master's thesis, COSIC Research Group at Department of Electrical Engineering, KU Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee (Belgium), 12 2012. Section 3.1.2, page 37.

[43] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. *IEEE Security and Privacy Magazine*, 2004:272– 282, 06 2004.

[44] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 21–41, Berlin, Heidelberg, 2013. Springer-Verlag.

[45] Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. Protected software module architectures. In *Securing Electronic Business Processes (ISSE'13)*, pages 241–251. Springer, 2013.

[46] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[47] Symantec. W32.stuxnet dossier. Security Response, February 2011. Available at `https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[48] Seehwan Yoo, Hyunik Kim, and Joongheon Kim. Secure compute-vm: Secure big data processing with sgx and compute accelerators. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, pages 34–36, New York, NY, USA, 2018. ACM.

[49] Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv.*, 44(3):17:1–17:28, June 2012.

Appendix A

Source Code and Installation

A.1 Latest Version Information

Please notice that Sancus is free software. The latest source code of the project is being actively maintained on GitHub. Specifically:

- The Sancus version provided with secure DMA support can be found at `https://github.com/S3rg7o/sancus-core`.
- Some examples written in C, leveraging SMs security guarantees can be found at `https://github.com/S3rg7o/sancus-core`. Here, the code for of some DMA attack examples is reported, too.
- Finally, installation directives, together with the source code and binary packages for the official Sancus release are respectively maintained at `https://github.com/S3rg7o/sancus-main` and `https://github.com/S3rg7o/sancus-support`. The software toolchain includes the simulator, compiler and runtime support libraries. All the provided binary packages have been tested on Ubuntu 18.04 LTS, but should work on any Debian-based GNU/Linux distribution.

A.2 Installation Instructions

Listing A.1: Installation instruction for the latest Sancus release. The version supports a secure DMA interface, in the sense that every Direct Memory Access to protected memory is detected and prevented by the MAL inside the core.

```
# 0. Pull the latest repository from the GitHub page
 git clone https://github.com/S3rg7o/sancus-main.git
 cd sancus-main

# 1. Install prerequisites
 sudo make install_deps     # default installation directory for
                            # Clang and msp430-elf-gcc is /usr/
                                local

# 2. Clone relevant Sancus project git repositories
 make

# 3. Build and install Sancus toolchain
 sudo make install    # Override default security level
                      # (64 bits), use SANCUS_SECURITY=128
                      # SANCUS_KEY=deadbeefcafebabec0defeeddefec8ed
                      # Default installation directory: /usr/local
                      # Set with: SANCUS_INSTALL_PREFIX=dir

# 4. To remove temporary files, after the installation type:
 make clean
 make distclean

# 5. To uninstall Sancus from the system launch the following from
     the sancus-main folder
 sudo make uninstall
```