POLITECNICO DI TORINO

Master degree in Electronic Engineering

Master Thesis

Defeating Hardware Trojans trough on-chip Software Obfuscation

Supervisor
Ernesto Sanchez

Candidate
Luca Sasselli

April 2018

Contents

List of Figures

List of Tables

Abstract

Due to the competitive nature of the embedded-system market, and the ever-increasing complexity of integrated circuits, most manufacturers prefer to outsource fabrication to third-party companies, to reduce costs and time-to-market. This however raises concerns on the trustiness of all parties involved in the production-chain, since opens the circuit to the threat of hardware piracy. Recently, Hardware Trojans have become a serious concern for embedded-system producers: an unknown attacker could modify the circuit during fabrication for malicious purposes, such as reducing its life-span, compromise high-level software security or allow remote access to sensitive data. A Trojan lays silently in the compromised circuit waiting for a rare sequence of events to carry out the attack, avoiding detection in post-production testing. This type of threat is extremely menacing since the presence of the malicious circuit is concealed until the attack is performed. Furthermore, an Hardware Trojan could be used to aid a software attack, thus moving its complexity from hardware to software, making its presence harder to spot in side-channel analysis. This thesis proposes an on-chip instruction obfuscator aimed at reducing the controllability of malicious elements hidden in the circuit trough software. Using a transparent design the instruction sequence is modified to eliminate Trojan activation sequences, while ensuring the correct execution of the original program. This technique requires minimum alterations to a pre-existing processor design and introduces a reasonable area delay overhead. The proposed approach was implemented in OR1200 processor, and tested with the Mibench benchmark, to verify its feasibility.

Chapter 1

Introduction

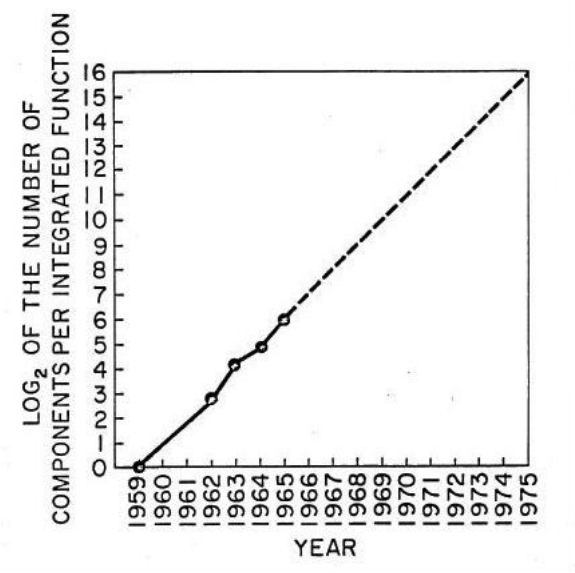

Figure 1.1: Original Illustration of Moore's Law, Electronics magazine (1965)

1.1 The threat of Hardware piracy

In 1965 Gordon Moore, Research director at Fairchild Semiconductor and later Intel co-founder, made an empirical observation on the state on *Integrated Circuits* (IC) at the time: it appeared that the number of transistors that could be integrated in a unit of area, doubled approximately every two years [1]. This statement, that would later become known as "Moore's law", predicted the same trend for the future, prophesying such wonders as "home computers".

The industry desire for smaller transistor have turned Moore's observation into a self-fulfilling prophecy: transistors have the unusual quality of getting better as they get smaller; a small transistor can be turned on and off with less power and at greater speeds than a larger one. This trend is in great part responsible for the enormous growth in the capability and the subsequent ubiquity of digital integrated circuits in the last 50 years. While many have predicted its downfall, Moore's law is still extremely relevant even today.

Unfortunately, as transistors get smaller, manufacturing becomes increasingly more challenging and expensive: it's estimated that by 2020 the upfront setup-cost for the smallest transistor size will be 20 billion US dollars [2]. For this reason, today, most IC manufacturers prefer to focus on designing Intellectual Properties (IPs) and outsource manufacturing to third-party companies, in order to meet the demand of the ever-growing electronic market. This strategy, known as *fabless manufacturing*, allows not only to drastically reduce the cost of production, but also to achieve a shorter time-to-market.

While consolidated by the financial success of many companies, this approach raises concerns on the trustiness of all the parties involved in the manufacturing chain, since opens-up the design to the threat of hardware piracy. An unknown actor, called *adversary*, could alter the system design before production for malicious purposes, unbeknown to the designer. This type of attack is most commonly known as *hardware Trojan Horse* (HT): the alteration to the IC could allow the attacker access to sensitive data, gain remote control of the system or simply create a critical failure; any type of high-level software security could be worthless if the underlying chip is compromised. These vulnerabilities have raised concerns regarding possible threats to military systems, financial infrastructures and even household appliances. HTs are designed to lay silently in the target system, to avoid detection, waiting for a *trigger*, usually a rare event known only to the attacker, to activate their malicious *payload* and carry out the intended attack.

To avoid alterations to the design during manufacturing, whether intentional or not, companies rely on post-production logic testing and verification. However, since HT are designed to be activated only by a rare event, and purposely engineered to conceal their presence, its unlikely that conventional test patterns could highlight the presence of this type of threats. New strategies should be devised to develop ICs capable of actively preventing HT insertion, by making such attack extremely difficult, if not impossible to perform.

1.2 Objectives

In this thesis work, a novel solution to the threat of Hardware Trojan Horses, targeting processors in embedded-systems, is proposed and evaluated. This approach relies on an on-chip instruction obfuscator to modify executed instruction at run-time, using

a difficult to predict pattern, in order to mitigate activation of triggers that rely on instruction sequences to be enabled, such as the one proposed by Yang et al. [3]. The main idea behind the proposed technique is to reduce the controllability over the executed instructions without modifying the actual program functionality; consequently, an attacker cannot reliably activate a malicious trigger via software.

Evaluating the effectiveness of this approach requires to design a functional instruction obfuscator that can be added to an existing processor design. Such device should allow a strong obfuscation scheme, without requiring excessive overhead or alterations to the target processor, in order to be a viable solution.

Firstly, the design requirements and challenges posed by a similar device and obfuscation pattern should be evaluated, in order to choose the most suitable approach that meets the requirements. Secondly, a proof-of-concept obfuscator should be designed for a processor, to serve as testbench to evaluate the effectiveness of the technique.

1.3 Contributions and Achievements

Performing software obfuscation directly on-chip requires employing strategies simple enough to be implemented directly within a processor core, with minimum overhead on its design, yet effective enough to eliminate the instructions used to activate the Trojan. The first contribution of this thesis is the development of a simple rule based obfuscation technique, that substitutes instructions with equivalent sequences of operations, as well as the creation of a set of such substitutions for the ORBIS32 instruction set. This required the development of design rules and requirements, as well as verification techniques, to ensure such equivalence.

The main result obtained is the HDL implementation and subsequent verification, of a proof-of-concept obfuscator design in an OR1200 open source processor. The obfuscator can be added to the original design with minimum effort, while introducing a reasonable overhead on area and delay. Experimental analysis proved that an on-chip obfuscator could be a viable technique to mitigate hardware Trojan activation.

1.4 Chapters overview

This thesis work is organized as follows. In the next chapter is presented an exhaustive background on hardware trojans, their taxonomy, state-of-the-art detection techniques and some attack examples. Chapter 3 defines the threat model chosen for the research, its limitations in the context of hardware security, as well as the development environment chosen for the research. Chapter 4 presents a detailed analysis of the obfuscation pattern and a theoretical model of its performance. Chapter 5 deals with

the design of the on-chip obfuscator, while chapter 6 presents the actual working implementation developed during this research. Chapter 7 describes the experimental results obtained, as well as, the techniques employed to obtain them. Chapter 8 states the conclusions drawn from the thesis work and proposes future activity on the topic.

Chapter 2

Literature survey

2.1 Hardware Trojan Horses

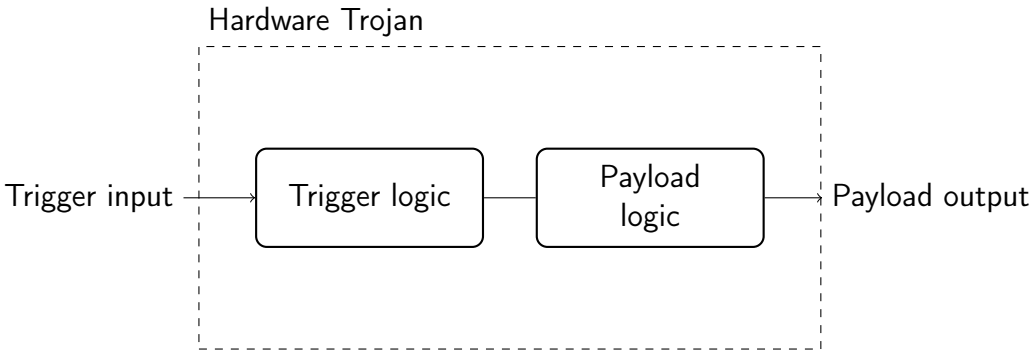

Figure 2.1: Simplified model of a Hardware Trojan Horse.

A Hardware Trojan Horse is a malicious alteration of an IC, inserted by an adversary in the design before or during the manufacturing process. Similarly, to their software counterparts, HTs conceal their behavior in order to lay unnoticed, and they only activate when a set of conditions are met. An HT is composed by a *payload*, the circuit that implements the malicious behavior, and a *trigger*, the circuit that detects the condition to deploy the payload.

2.1.1 Design challenges

Evading detection is a critical characteristic in designing an effective Trojan: the HT must not only avoid being activated by test patterns, but also evade detection from visual and side-channel inspection. An HT must be small enough to be inserted in the empty space of the placed and routed chip layout handed to the manufacturer, must have a small power consumption to avoid producing unexpected power fluctuations,

and must produce a negligible timing perturbation in order not to affect timing constraints. Trojan trigger must be both controllable, in order for the attacker to activate it at will, and complex, to avoid being activated accidentally. These characteristics make designing an HT trigger a trade-off between complexity and area.

2.1.2 Insertion points

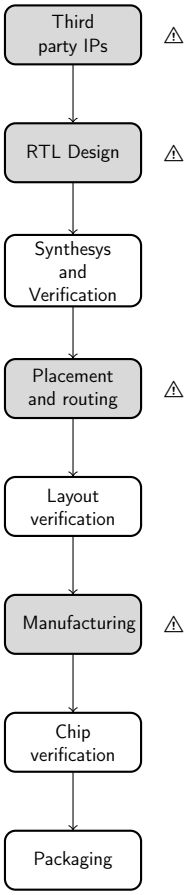

Figure 2.2: HT threats during IC design cycle.

The design cycle of an IC offers multiple opportunities for an attacker to insert an HT in the design. The insertion points can be divided into *Design time* and *Fabrication time* attacks. During design, a first threat is posed by *Intellectual Properties* (IP) from untrusted third-parties. These are basically treated as black-boxes by the system designer, that relies solely on the specifications provided by the IP designer. This represent already a possible threat to the IC security, since the IP could hide malicious functions. Other than IP, a rogue member of the design team could insert a hardware Trojan for unknown reasons during design. During fabrication, the place and routed

design could be manipulated immediately ahead of manufacture by the attacker, to add new logic or modify the existing one. Other than that, even if the IC mask layout is unaltered and conforms to the designer expectations, the attacker could insert a Trojan by altering the doping of selected regions.

2.1.3 Trojan in microprocessors

The presence of a processor in a design allows moving complexity from hardware to software, with the advantage of increased flexibility for the designer. However, this flexibility can also be exploited by the attacker to implement malicious functionalities. A malware could, aided by an HT, perform an attack normally impossible due to high level software security, allowing to simplify the payload and reduce the Trojan overhead on the design.

2.2 Taxonomy

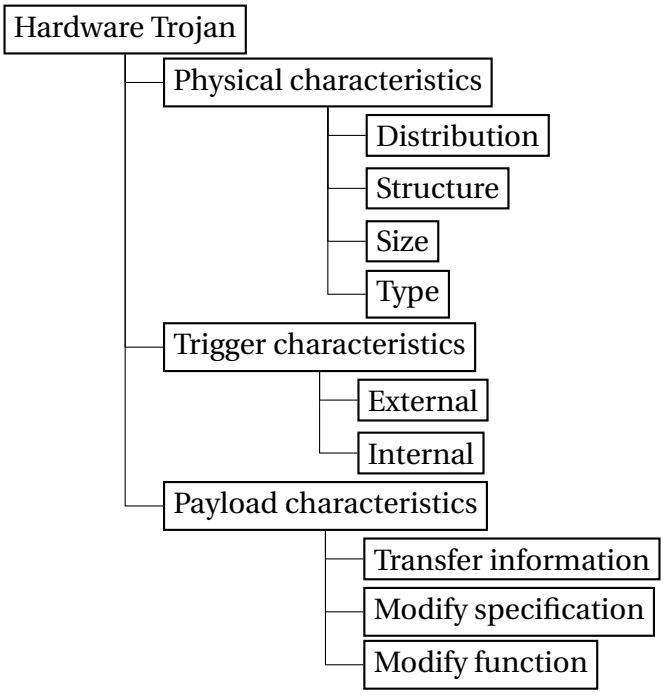

Figure 2.3: Detailed Hardware Trojan Horses taxonomy.

Wang, Tehrabipoor and Plusquellic developed the first detailed taxonomy for hardware Trojans[4]. This classification divides HT by their physical characteristics, trigger behavior and payload action.

HTs are classified in the *physical* category by their hardware manifestation, according to their distribution, structure, size and type. *Distribution* describes HT by their location on the chip layout. *Structure* divides the insertion type whether the adversary was forced to regenerate the physical layout or not. Layout regeneration could result in different placement of some components, potentially alerting the designer of a possible hardware attack. *Size* categorizes not only the actual area of the HT, but the extent of the alterations performed on the original layout. Finally, *type* characterizes Trojans by their implementation, that can be either functional or parametric. The former means that a simple logic circuit is realized within the design through the addition or deletion of logic gates or transistors, the latter refers to Trojan that are created by altering existing logic or wires.

The *trigger* category classifies the activation signal that causes the payload to become active, and can be either internal or external. *Internal* triggers are activated by internal signals and can be further divided in *always on* and *conditional*. "Always on" are triggers that don't require activation and can carry out the attack at any time, "conditional" are triggers that require a specific activation condition to start. External triggers are activated by interaction with the outside world, by means of a sensor or an antenna.

In *payload* category HT are classified by the type of attack carried-out when the trigger is activated. The action can be of three types: transmit information, modify specification and modify function. The *transmit informations* class refers to HT that leak confidential information to the adversary. *Modify specification* refers to Trojans that change the chip parametric properties. Finally, the *Modify function* class, groups the HTs that change circuit function by adding logic or by removing or bypassing the existing one.

2.3 Attack examples

To better understand the threat posed by HTs in this section presents a collection of attacks that could be carried out by means of hardware piracy. This collection isn't intended to be an exhaustive one, but to present the reader some real-life attack scenarios.

2.3.1 Denial of Service

In *Denial of Service* (Dos) attacks the attacker prevents the user from accessing the services provided by the system by blocking access to it or creating a critical failure. Destructive DoS attacks are especially dangerous since they don't require complex operations to be performed, but they simply break the system from the inside. Becker et al. presented an extremely stealthy approach to this type of attacks [5]: by selectively

modifying doping in selected regions of the circuit the attacker could artificially produce the effect of aging, thus increasing the unreliability of the target. Dopant level attacks are impossible to detect optically since they don't alter the layout of the IC.

2.3.2 Privilege escalation

In a computing system, there's no worst attack scenario than an untrusted program gaining elevated privileges without the user consent. While high level software security is designed to prevent such scenarios from occurring, this could be easily circumvent with the introduction of a hardware Trojan. Yang et al.[3], presented a fabrication-type attack that targets the privilege bit of an OR1201 processor. This bit is set to 1 when a program operates in supervisor mode, allowing the program unrestricted access to the processor configuration registers, memory management and peripherals. Rare signals are used to activate an analog trigger that uses capacitive coupling to accumulate charge in a capacitor that acts as a makeshift counter. When the charge reaches a predetermined threshold level, the privilege bit is set from 0 to 1, allowing the program that has issued the signal complete access to the processor. The use of an analog Trojan has the advantage of being extremely small but the disadvantage of being susceptible to process variations.

2.3.3 Side-channel attack

Side channel information, such as timing or power consumption, is frequently used to break cryptography: the attacker can use statistical analysis on this data to determine intermediate results of cryptographic computations, even on an untampered but poorly designed system. An HT could be used to compromise encryption by amplifying side channel-information or by directly leaking secret keys or unencrypted data through it, where can be captured by the attacker for malicious purposes [6]. To avoid detection, these type of Trojans use either information from the data cache or specific instruction sequences to be triggered.

2.4 Detection techniques

After manufacturing, in order to ensure that a produced unit has not been tampered by the adversary, the authenticity of the chip must be checked. This means performing a set of tests in order to ensure that the unit performs only the functions originally intended by the designer. The set of post-manufacturing steps required to validate a chip conformance to the original design is called *silicon design authentication*[7] and the device tested is the *IC under authentication* (IUA). Authentication must not only be reliable, but also simple enough to be applied to the entire production. Standard VLSI fault detection tools, such as *automatic test pattern generation* (ATPG), are ineffective

against HT since they rely on the original untampered design to generate test patterns: without knowledge on the Trojan logic and its relation to the original logic of the chip, test vectors that activate the trigger cannot be produced. Moreover, even if a HT is activated by chance, it's only possible to detect attacks that modify the internal states or outputs, and not attacks that leak informations through side channels (e.g., the power supply). Trojan detection is thus challenging for two main reasons: one it's activating the Trojan, the other is detecting the attack.

Over the last few years several Trojan detection methodologies have been developed, and they can be categorized into three families: *side channel analysis*, *Trojan activation* and *architecture level*. Furthermore, it exists an entirely separate class of destructive techniques. In these the IC is physically opened and each layer is reverse engineered to extract all components, that are later analyzed to detect tampering. This approach, while extremely effective in detecting Trojans, is time-consuming, expensive and cannot be applied to the entire production, since it destroys the IC. Nevertheless, it can be used to highlight presence of hardware piracy in the manufacturing process.

2.4.1 Side channel analysis

Side-channel signals can be used to detect the presence of an HT within an IC. The insertion of a Trojan in facts alters the parametric characteristics of the IUA with respect to the expected behavior. HT presence can thus manifest as degraded performance, increased unreliability and different power characteristics. This type of authentication approach requires first analyzing a Trojan-free device called *golden chip*. The golden chip can be either manufactured by a trusted company, or sampled from the production, assuming that the adversary has inserted Trojans only in a limited batch of the produced ICs. This last approach requires identifying Trojan free units, by means of extensive testing of randomly selected dies. Side channel method are divided into two categories: *power-based* and *timing-based*. Power-based techniques use side channel information to detect the HT contribution to the IC power consumption. Timing-based method uses delays on selected paths as fingerprints, trying to find differences in delay between a IUV and golden attributable to a Trojan insertion.

Both of these methods have the disadvantage that they search the HT when the payload is not active: in this state HT contribution to parametric characteristics could be comparable to standard process variation, thus rendering Trojan presence testing unreliable or even impossible.

2.4.2 Trojan activation

Trojan activation methods try to accelerate Trojan detection by trying to activate the payload, in order to make the HT more visible in side channel analysis or even detect

the attack itself. HTs can be triggered by stimulating networks that, in normal IC operations, are usually rarely activated. These signals are most likely to be used by the adversary in the HT trigger design. These techniques are divided in *region-free* and *region aware*. Region free techniques do not target specific areas of the circuit, but the entire circuit as a whole. Region-aware instead focus on separate partitions of the IC, by trying to simultaneously increase the activity in a partition and minimize it in all the others, in order to reduce noise on side channel measurements. However, if a trigger relies on signals belonging to separate regions of the circuit the last techniques could be less effective than the first.

2.4.3 Architecture level

Architecture level techniques are capable of detecting Trojan activation by monitoring the behavior of the IC on the field: if a misbehavior is detected, some corrective actions could be implemented to prevent the attack completion. This type of techniques has the disadvantage that cannot prevent *Denial of Service* type attack and requires additional hardware to be implemented. Moreover, since it's embedded in the IC, it vulnerable to hardware piracy. This eventuality can be mitigated by designing the hardware with the paradigms proposed in the next section.

2.5 Design for hardware trust

Similarly to what has been done with paradigms such as *design to manufacturability* (DFM) and *design for testability* (DFT), the idea is to modify the design flow of an IC to include authentication features to the design. These features are included with the goal of aiding Trojan detection with side channel techniques, or mitigate attacks by means of hardware isolation.

Moving components in the design to insert a Trojan has the effect of altering the parasitic parameters: this can be detected and magnified by means of *physically unclonable functions* (PUF) [8]. PUF are integrated structures that perform a function highly dependent on the process variation and are thus highly affected by design alterations. After production PUF are analyzed to highlight modified ICs: these methods must be able to differentiate faults or standard process variation from intentional attacks.

Other techniques rely on the insertion of dummy flip-flops in nets with low transition probability, in such way that they don't alter nominal delay or behavior[9]. These flip-flops are used to force an increased switching activity on such nets in an effort to trigger HTs, or simply highlight their presence in side channel inspection.

Finally, another interesting approach is to allow reverse supply voltage for some gates in some safety critical regions of the IC[10]: if a four input AND gate is reverse powered it behaves like a NAND gate. This allows to switch the probability of having

the gate output as 1 from 1/16 to 15/16, thus forcing a rare condition to be extremely frequent.

Chapter 3

Case study

3.1 Threat model

The proposed threat model closely resembles the one described in [3]: a processor is compromised by the attacker during fabrication, by injecting an HT. The trigger monitors wires and state within the CPU and activates the attack payload under very rare conditions, such that the HT lays silently during normal operation and test procedures. The signals monitored are stimulated by using a *precise sequence of instructions* executed by a malware program: the attack can be either carried by the HT alone, or by the program itself aided by the Trojan. To perform the attack, the malware, that wouldn't be harmful in an untampered system, executes a precise sequence of instructions enabling the trigger; the sequence is chosen by the attacker to be rare, to prevent accidental activation, and short, to simplify the trigger logic.

This threat model is interesting for multiple reasons:

- Instructions have a high controllability;
- The attacker can estimate the rarity of a given trigger sequence by studying other programs and compilation patterns;
- The attack can be carried by the program, aided by the HT, thus simplifying the payload. This allows to either reduce the Trojan footprint altogether, or increase the complexity of the trigger while remaining stealthy.

3.2 Proposed solution

The proposed solution consists in an on-chip software obfuscator that modifies the sequence of instructions executed by an embedded processor, effectively altering the predefined sequence of instructions used to trigger the injected HT. The obfuscation pattern employed depends on parameters imposed after manufacturing by the CPU

user: while the attacker is aware of the presence of the obfuscator, he/she cannot control the instructions executed, since these depend on the algorithm configuration. The obfuscation is thus non-deterministic from the point of view of the attacker, but deterministic for the end-user.

3.2.1 Why an on-chip obfuscator?

An on-board obfuscator offers multiple advantages, as well as disadvantages, with respect to a purely software approach. In theory, if obfuscation represent an effective solution to mitigate HT activation, the same results could be obtained by obfuscating untrusted software before executing it on the target machine. In most embedded systems, software is usually designed by the system designer or by a third-party developer, and the embedded firmware is rarely modified during the product life-cycle. The designer can use software obfuscation techniques to ensure that the software has not been modified for malicious purposes, in this case activating a Trojan. However, in more complex systems the user can run custom applications from third-party developers, e.g. mobile applications, programs, client-side scripts, thus the designer has limited control over all the code executed on the machine, and must rely on high-level software security. An embedded obfuscator ensures that the attacker has limited control over the machine instructions executed in the processor, regardless of the context in which the embedded processor operates.

3.3 Limitations

3.3.1 Overhead

An on-board solution implies an area, as well as delay overhead on the design: to be a viable solution, an embedded obfuscator must minimize both. This can be done by selecting an obfuscation technique simple enough to implemented directly in hardware with minimum impact, yet effective enough to provide improved security with respect to a system without obfuscation. Moreover, the obfuscator design must be also flexible to be easily inserted in an existing processor design, and avoid redesigning from scratch the entire system.

3.3.2 Piracy threats

As with most architecture-based solutions, an on-chip obfuscator could be potentially compromised by an HT: if the Trojan is inserted during manufacturing the attacker could be aware of the presence of the obfuscator and modify it. This can be prevented by designing the obfuscator to be authenticated and tested after production: if it is

assumed that obfuscation prevents the proposed attack from being performed, the attacker must always modify the obfuscator to insert an HT. Verifying the authenticity of the obfuscator alone is cheaper and faster than performing the same task for the entire system.

Moreover, the attacker could still carry out a successful Trojan activation by designing an instruction sequence that is obfuscated in a predictable way, or exploiting faults in the obfuscator logic.

3.3.3 Obfuscation efficiency

As in the case of a purely software approach, obfuscation must be capable of ensuring the destruction of sequences used to trigger the HT. This, while a critical requirement in both a software and hardware obfuscation, is especially problematic in the latter. Software obfuscation can operate on a program before execution, exploiting its logic to generate code that is very different from the original one, and can be finely tuned to maximize obfuscation. A hardware approach however has limited knowledge on the program executed by the processor, and must operate with limited resources while being equally effective.

3.4 OR1200 processor

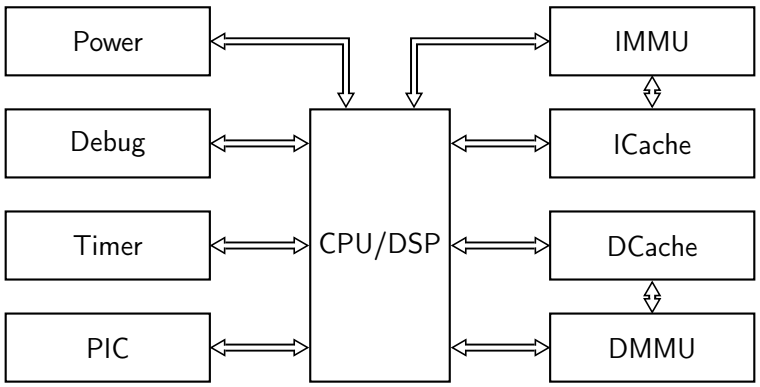

Figure 3.1: OR1200 architecture

The proposed on-chip obfuscator was developed for the *Open RISC 1200* (OR1200) soft processor [11], a synthesizable open-source CPU core developed by *OpenCores*, an online community dedicated to open-source IPs. This section will introduce some features of this processor as well as the reasons for its choice as development platform for this project.

3.4.1 Overview

The OR1200 is an open-source CPU core with *Open RISC 1000* (OR1K) Harvard architecture, implemented in Verilog HDL. It uses the 32-bit *Open RISC Basic Instruction Set* (ORBIS32) and optionally the *Open RISC Floating Point eXtension* (ORFP32X), to add IEEE-754 compliant single precision floating-point support. The instruction set has 5 types of instructions and supports direct, and register displacement addressing. The pipeline is composed by 5 stages capable of executing most instructions in a single clock cycle. Other features include a MAC/DSP unit, a debug unit for real-time debugging, programmable interrupts and power management. It uses a Wishbone B3 compliant bus interface and is thus compatible with a broad number of open-source peripherals. An overview of the processor is shown in figure 3.1.

3.4.2 CPU/DSP

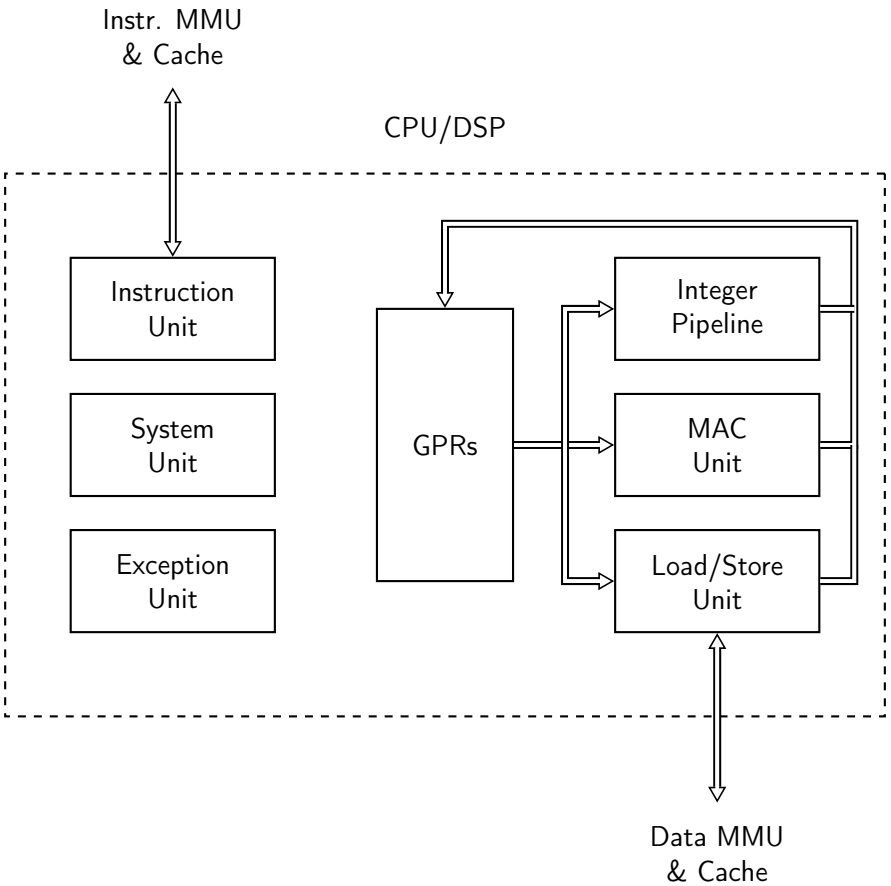

Figure 3.2: OR1200 CPU/DSP block diagram

The central processing unit of the OR1200 is composed by the following elements:

- Instruction unit
- GPR
- Load/Store unit
- Integer pipeline
- MAC unit
- System unit
- Exception unit

Instruction unit The instruction unit implements the basic instruction pipeline. It is responsible of retrieving the correct instruction from the memory subsystem and dispatch it to an available execution unit. Both conditional branches and unconditional jumps are managed by this component.

GPR The OR1200 has 32 32-bit general purpose registers. The file system is implemented has a dual-port synchronous memory.

Load/Store unit The LSU manages all the data transfers between GPRs and the data bus. It is implemented as an independent execution unit, thus stalls in the memory subsystem affect the pipeline only if a data dependency exists.

Integer pipeline The integer execution pipeline manages the execution of arithmetic, compare, logical and shift-rotate instruction. Most of these instructions can be executed in a single clock cycle.

MAC unit To add basic DSP functions, the OR1200 implements a *multiply and accumulate* unit (MAC) with a 48-bit accumulator. The MAC unit is fully pipelined and can execute a new operation each clock cycle.

System unit The system-unit manages all the signals unrelated to instruction/data interfaces, and contains all the special-purpose registers.

Exception unit The OR1200 has multiple exception sources: external interrupts, some memory access conditions, internal errors, system calls and internal exception. When an exception occurs the exception unit passes control to a handler at a predefined memory location, depending on the type of exception. Exceptions are handled precisely, thus all the instructions executed before the event are valid.

3.4.3 Caches and MMUs

The OR1200 is a Harvard architecture processor, and thus has independent caches and *Memory Management Units* (MMUs) for instructions and data. Both these units are implemented identically regardless the type.

Cache In the default configuration of the OR1200 cache is 8KB (alternatively 1KB, 2KB or 4KB), 1-way direct mapped. It implements the last-recently used replacement policy.

MMU MMU allows the OR1200 to implement a virtual memory scheme, providing access protection and effective to physical address translation.

3.4.4 Other subsystems

Power management To reduce power consumption the OR1200 provides three low-power modes, that can be used to dynamically turn on/off internal modules. *Slow mode* uses optimized clock dividers to reduce consumption at the cost of a reduced clock frequency, and can be controlled via software. In *Doze mode* software processing is suspended and clocks to internal modules is gated; only tick-timer is active. In *Sleep mode* all the modules are disabled and clock gated. The CPU leaves Doze/Sleep mode when an interrupt occurs.

Debug interface This unit aid developer in debugging code. It supports only a subset of the debug features specified by the OR1K architecture such as breakpoints, watch-points and real-time trace.

Tick-timer A tick timer facility is present to precisely measure time and schedule system tasks. The timer operates on an independent clock source, can count up to 2^{32} ticks, has makable interrupts and can operate in single-run, restartable and continue mode.

PIC The *Programmable interrupt controller* (PIC) is responsible for processing external interrupts and forwarding them to the CPU has high/low priority. It has 32 independent input, 30 of them maskable and with configurable priority.

3.4.5 Tools and Software

The Open RISC project has spawned a wide variety of tools that make this processor an extremely interesting choice. In the following are reported the tools used in this project.

FUSESoC

The *Open RISC Reference Platform System-on-Chip* (ORPSoC) project is a complete SoC based on the OR1200 processor, used as reference implementation for the Open RISC processor family. It provides a set of predefined configurations for memory, interrupt and peripherals, as well as build and simulation tools, to provide a starting point for developing an Open RISC based SoC. Within this project, Olof Kindgren, created *FUSESoc*[12], a HDL package manager, created with the goal of increasing reuse of open IP cores in different projects. FUSESoC allows to build an HDL system by composing modules from a wide library of processors and peripherals, streamlining simulation and FPGA build in a reusable environment. Designed initially solely to aid the creation of Open RISC systems, is now a completely independent project.

GNU toolchain

The GNU toolchain is a collection of programming tools developed for the GNU project. These programs are essential tools for software development, and are widely employed in both open-source and proprietary software. A version of the GNU toolchain compatible with the OR1K architecture have been developed by OpenCores, allowing to compile almost any software for the OR1K processor. Moreover, it allows to use the debug interface of the OR1200 to debug applications on an ORPSoc system via JTAG. The existence of an OR1200 compatible GNU toolchain has greatly contributed to the selection of the OR1200 for this project.

Newlib library

A port of the Newlib library was also developed by OpenCores. Newlib[13] is C standard library designed to operate in bare-metal applications (without an operating system), designed for embedded systems. This allows building fully functional software for an ORPSoC system, without "reinventing the wheel".

Chapter 4

Obfuscation

4.1 Introduction

Obfuscation, in Computer Science, is the deliberate process of making code obscure or unclear, in an effort to prevent reverse engineering of a program for malicious purposes. These techniques are common in the software world, to modify either source code, instructions or metadata, without changing the program final outputs [14]. In the proposed approach, instruction obfuscation is used to mitigate HT activation by eliminating trigger sequences with an on-chip obfuscator. While this is very similar to traditional software obfuscation, the end goal is simpler: the objective is not concealing the purpose of the program, but only eliminating trigger sequences used to activate the trojan.

In order to perform obfuscation directly within the processor, the obfuscation mechanism must be, while effective, also simple enough to reduce the overhead introduced by the additional hardware. According to the described threat model, trigger sequences are precise pattern of instructions recognized by the HT. The goal of the obfuscator should be removing these patterns by altering them to the point that they are no longer recognized by the HT as trigger sequences. The obfuscation must also be difficult to predict, otherwise the attacker can exploit repetitive obfuscation patterns to carry out the attack.

To summarize the key characteristics of the required obfuscation pattern are the following:

1. Simplicity
2. Strength
3. Low predictability

4.2 Proposed technique

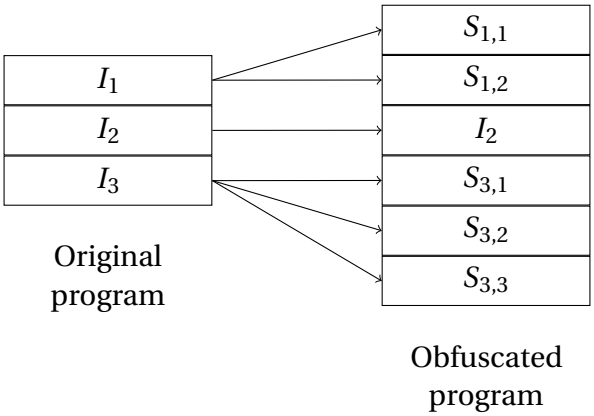

Figure 4.1: Proposed obfuscation mechanism

The proposed obfuscation technique replaces an instruction from the original program, here called *Reference Instruction*, with a *Substitution*, a sequence of operations that performs the same task, but differently enough to avoid being detected by the HT trigger. A sub-set of the instructions of the program is substituted at run-time: if an instruction of the concealed trigger sequence is substituted, the trigger is modified and potentially eliminated. A simple representation of the obfuscation technique is presented in figure 4.1: if the reference program is composed by three instruction, I_1, I_2, and I_3, if I_1 and I_3 are obfuscated, the same program will be $S_{1,1}$, $S_{1,2}$, I_2, $S_{3,1}$, etc., where $S_{i,j}$ is the $j-th$ instruction of a substitution for the reference instruction i. Instruction obfuscation is performed by an on-chip obfuscator unit, using a rule based approach: for each instruction in the instruction set, an array of possible substitutions is defined creating a substitution library. This library contains the set of rules required to convert the reference instruction into the actual executable code: at run-time, the obfuscator replaces an instruction with one of its substitutions chosen according to a predefined algorithm configured by the user after manufacturing. Since this technique only obfuscates one instruction at the time, it can be inserted into an existing processor pipeline with minimum effort.

This approach allows to satisfy all the requirements presented in the previous section. In terms of *Simplicity*, the obfuscations doesn't require any knowledge about the code executed, since all substitutions are exactly equivalent to the instruction that they substitute: this increases the substitution design effort, but simplifies the resulting hardware. *Strength* is ensured by designing substitutions that are "different" enough from the original instruction, and by substituting enough instructions to increase the

probability of trigger elimination. Finally, *Low predictability* is granted by implementing the obfuscation pattern with a suitable algorithm, and a sufficiently large substitution library.

4.2.1 Design challenges

Reference/Substitution equivalence

Designing substitutions, is a challenging endeavor, since they must guarantee absolute equivalence with the reference instruction: if a substitution behavior differs only slightly from the reference instruction, execution could be heavily or only slightly affected. Regardless, the obfuscator would compromise the processor rendering it unreliable, and thus useless. In that case, the misbehaving instruction could be extremely difficult to identify. A suitable set of tests has to be designed to verify the equivalence between reference and substitution in a controlled environment.

Execution overhead

Since substitutions are, in most cases, composed by multiple instructions, a substitution will always take more clock cycle to complete with respect to the instruction that they obfuscate, thus the obfuscator will always increase the number of executed instructions. To limit the impact of substitution on run-time substitution length should be minimized when possible.

Area overhead

The substitution library is stored directly within the obfuscator, thus the hardware footprint is directly proportional to the number of substitutions implemented by the obfuscator designer. A suitable substitution encoding has to be designed to store substitution, in order to minimize the area overhead on the design.

4.3 Substitutions

Substitutions are sequences of operations that behave exactly like an instruction that they obfuscate, the reference instruction, regardless of its operands, its position within the code and the processor state. In order to do that, a substitution must adhere to the following rules:

1. It must produce the same result of the reference
2. It must set the same flags in the status register
3. It must not alter any register except the result destination (if any)

4. It must not alter any memory location except the result destination (if any)

5. It must not generate exceptions that are not produced by the reference

4.3.1 Types

Broadly speaking, while multiple techniques can be employed to design substitutions, during this thesis work two general strategies were identified, each one producing a conceptually different type of substitution. These types where called namely *True Substitutions* and *Dummy Substitutions*; this section will provide a detailed analysis of both.

True substitutions

True Substitutions are equivalent to the reference instruction by implementing the same logic with different assembly code. Here's an example of this type of substitution:

Listing 4.1: Example of True substitution

```
1 # Reference: l.movhi rD,I
2 l.xori rD,r0,I
3 l.slli rD,rD,16
```

The reference instruction `l.movhi` stores the immediate value 1 into the most significant half-word of the register rD, while the least-significant is cleared to zero. This operation can be similarly implemented in a two-step process by the proposed substitution: first the immediate value is stored into register rD as is, by xoring it's value with register r0 (which is set to zero by default), then the register is left-shifted the by 16-bit to move its content into the most significant half-word. This operation is perfectly equivalent to the reference.

Dummy substitutions

Dummy Substitutions contain the reference instruction "padded" with useless instructions, called *Dummy instructions*, that either produce no result, or do not alter the processor state. An example of true substitution is the following:

Listing 4.2: Example of Dummy substitution

```
1 # Reference: l.movhi rD,I
2 l.xor r0,r0,r0
3 l.movhi rD,I
4 l.sll r2,r2,r0
```

In this case, the reference instruction is implemented as is, but is preceded by a xor between null values, and succeeded by a left shift by 0: none of these two operations has any affect, and are only used to add some form of alteration to the reference instruction.

True vs. Dummy substitution

A true substitution produces a stronger obfuscation of the machine code, since the resulting code is completely different from the original instruction, but must be created by hand by the obfuscator designer, carefully analyzing the reference instruction behavior. Moreover, except some fortuitous cases, a true substitution is much longer than a dummy one. On the other hand, dummy substitutions can be created in a straight forward manner since uses always the reference instruction plus the useless ones, but they produce a repetitive obfuscation pattern that could be easily exploited by the attacker. Since not all the instructions can be obfuscated with true substitutions, a combination of both must be used to create a complete substitution library.

4.3.2 Design

Dummy substitutions

Table 4.1: Dummy instructions examples

l.and r0,rX,r0	**l.nop**
l.andi r0,rX,0	**l.or** rX,rX,r0
l.extbs r0,r0	**l.ori** rX,rX,0
l.extbz r0,r0	**l.ror** rX,rX,r0
l.exths r0,r0	**l.rori** rX,rX,0
l.exthz r0,r0	**l.sll** rX,rX,r0
l.extws r0,r0	**l.slli** rX,rX,0
l.extwz r0,r0	**l.sra** rX,rX,r0
l.ff1 r0,r0	**l.srai** rX,rX,r0
l.fl1 r0,r0	**l.srl** rX,rX,r0
l.movhi r0,0	**l.srli** rX,rX,0
l.mul r0,r0,r0	**l.xor** r0,rX,rX
l.muli r0,r0,r0	**l.xori** rX,rX,0
l.mulu r0,r0,r0	

Dummy substitutions are extremely easy to design, since they only "pad" the reference instruction with useless ones. A set of sample dummy instructions is proposed in table 4.1. The number of instructions used as padding and their position in the

substitution is completely up to the designer. However, since true substitutions are usually very long, using short dummy substitutions could allow to drastically reduce the average substitution length, and thus the overall impact of obfuscation. A possible approach could be using only a dummy instruction before or after the reference, alternating the position across the library.

Listing 4.3: Dummy substitution with padding after the reference.

```
1 # Reference: l.addi rD,rA,I
2 l.addi rD,rA,I
3 l.andi r0,r0,0  # Pad instr.
```

Listing 4.4: Dummy substitution with padding before the reference.

```
1 # Reference: l.addi rD,rA,I
2 l.andi r0,r0,0  # Pad instr.
3 l.addi rD,rA,I
```

Unfortunately, this type of substitution could lead to a weaker obfuscation: in fact, it was previously assumed that if an instruction of a trigger sequence is substituted, the trigger is eliminated, but using this type of substitution the assumption is no longer valid. If a 2 instruction dummy with padding before the reference is substituted to the first instruction of the trigger sequence, the trigger sequence is unmodified.

True substitution

Designing a true substitution is an extremely delicate process: excluding instruction that can't be implemented with true substitution by design (e.g. special purpose register access, branching instructions, etc.), each instruction has a unique behavior that must be re-implemented as a set of different operations. This means that there is no systematic technique that can be employed to write true substitutions, but there are some general strategies that can be reused in multiple instances:

- An immediate instruction can be substituted with the register version of the same operation by storing the immediate in a temporary register.
- Some instructions behave like others in some specific case that can be enforced (e.g. `l.add`/`l.addc`).
- Boolean properties can be used to implement an instruction with another (e.g. DeMorgan's law, masking, etc.)

To implement some substitutions, it may be required to store temporary results in the register file: unfortunately, since substitutions must work in any program, regardless of status, data or position, writing any register could mean damaging the program data. The only two registers that can be used for storage, even if only under some strict

conditions, are rD (the reference instruction destination register) and r0. In the OR1K architecture, as in most RISC architectures, register r0 is used to store the value 0, but it's not hardwired to this value: r0 is a regular register that must be manually cleared at reset. For this reason the OR1K manual forbids write operation on this register, to avoid changing its value. Since substitutions are atomic, this register can be used as temporary storage, provided that is cleared before the substitution has finished. Using these two register can be safely done only when they are not used as operands in the reference instruction. This can be easily ensured in the case of immediate operation: this type of instructions only uses one input register, and thus after copying its content in a known location (either r0 or rD), the other can safely be used as temporary storage.

4.3.3 Verification

Verifying the equivalence between reference and substitution in a critical task in designing a complete substitution library. Checking each substitution manually by inspecting the code doesn't guarantee absolute reliability. Let's take as an example a dummy substitution for instruction l.and:

Listing 4.5: Bad Dummy substitution

```
1 # Reference: l.and rD,rA,rB
2 l.addi r1,r1,0
3 l.and rD,rA,rB
```

At first glance this substitution looks a viable candidate: l.addi produces no result since it increments by 0 the content of register r1, and the reference is executed immediately after. By doing so, however, the carry-flag (CF) in the supervisor register is cleared, since the dummy instruction produces no carry: this doesn't happen in the original instruction, and thus this substitution could potentially break a program execution. Furthermore, using rD and r0 as temporary register can lead to unexpected errors:

Listing 4.6: Bad True substitution

```
1 # Reference: l.andi rD,rA,rB
2 l.xori r0,rA,-1    # Bitwise negation of rA
3 l.xori rD,rB,-1    # Bitwise negation of rB
4 l.or rD,rD,rA    # rD = rA' + rB'
5 l.xori rD,rD,-1    # rD = rD'
6 l.xor r0,r0,r0     # Clears r0
```

In this case, DeMorgan's law is used to implement l.and using l.or. This should work perfectly in theory, since after the substitution is performed the status register is unmodified and the result produced is the same. However, the substitution designer doesn't know the actual register for rA, rB and rD, but the substitution must work regardless of their value: in the proposed example, if rB is set to r0, the substitution produces a wrong result.

In order to ensure that all the reference/substitution equivalence criteria are met, a test environment has been written in Python to automatically test the entire library. For each substitution two test programs are executed: one to check the result, and one to check the supervisor register. In both these tests exceptions are logged to spot unexpected events. The test environments automatically creates and builds the test programs and compares the results to highlight errors. Each test is repeated multiple times with random operand values: this ensure a reasonable level of confidence about the substitution equivalence.

Listing 4.7: Pseudo-code of the substitution library test

```
1 test_substitution(ref, sub){
2
3   // Test result equivalence
4   build_result_test(ref, sub);
5   run_test();
6   if(!check_results()) return false;
7   if(!check_exceptions()) return false;
8
9   // Test status register
10  build_status_test(ref, sub);
11  run_test();
12  if(!check_status()) return false;
13  if(!check_exceptions()) return false;
14
15  // All tests passed!
16  return true;
17 }
```

4.3.4 Strength

As already detailed in the threat model, instruction sequences are only a mean to activate the HT trigger, and are not actually "seen" by the trigger itself. What triggers the Trojan is a sequence of values on some internal signals stimulated using the instruction sequence. So, even if a substitution removes a reference instruction from the sequence, the obfuscated code could still produce similar internal signal values and be recognized as the trigger sequence. Some sort of score is required to evaluate, during design, the degree of difference between a substitution and its reference instruction.

To evaluate the strength of a substitution, the operand independent output signals of the decode stage produced by the reference instruction, are compared to the ones produced by the substitution code in terms of average *Jaccard* and *Simple Matching Distance* (SMD). These signals have been selected since they only depend on the type of instruction decoded and thus have high controllability.

Given two strings of bits x and y, Jaccard distance, D_{jacc}, and Simple Matching

distance, D_{smd}, are defined as:

$$D_{jacc}(x,y) = \frac{M_{01} + M_{10}}{M_{01} + M_{10} + M_{11}}$$

$$D_{smd}(x,y) = \frac{M_{01} + M_{10}}{M_{00} + M_{01} + M_{10} + M_{11}}$$

where M_{ij} it's the number of bits whose value changed value from i to j from string x to y. Both values are the ratios between bits changed between sequences, and total bits, the only difference between them is that SMD considers M_{00} in the total, while Jaccard distance doesn't, and thus gives a more optimistic distance index.

Using this two distances two scores can be defined:

$$S_{jacc}(r,s) = \frac{\sum_{i=0}^{n} D_{jacc}(r,s_i)}{N}$$

$$S_{smd}(r,s) = \frac{\sum_{i=0}^{n} D_{smd}(r,s_i)}{N}$$

where r is the decode word of the reference instruction, and $s = \{s_0, s_1, ..., s_n\}$ it's the set of decode words produced by each instruction in the substitution.

4.4 Obfuscation performance

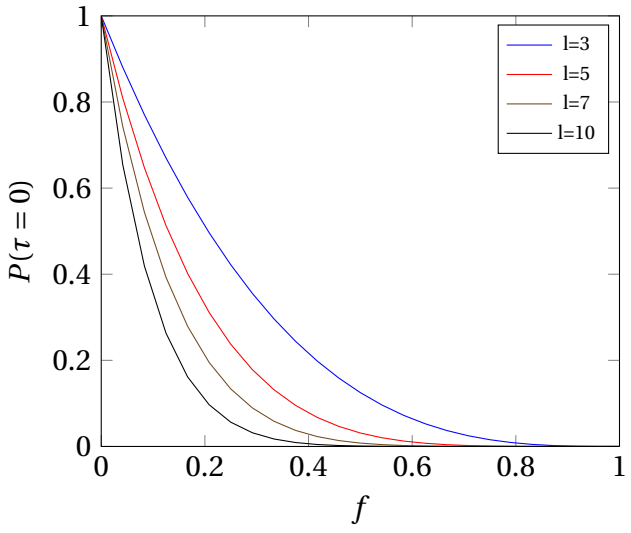

Figure 4.2: Relationship between f and $P(\tau = 0)$

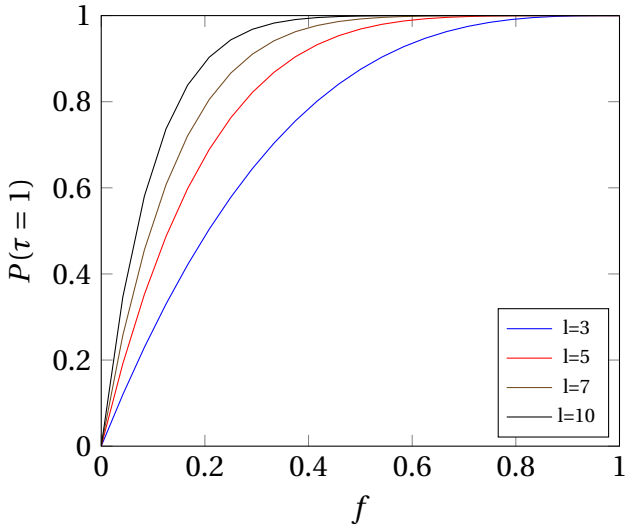

Figure 4.3: Relationship between f and $P(\tau = 1)$

Obfuscation efficiency can be evaluated by studying the probability of breaking a trigger sequence, and by the program execution overhead produced. Since no assumptions can be made on the position of the trigger sequence within the code, the obfuscator chooses the instructions to obfuscate randomly. The designer can decide with which frequency a substitution in performed: this parameter is called *Substitution Frequency* (f), and is the ratio between obfuscated instructions per operations performed.

4.4.1 Efficiency

A simplified efficiency model can be created assuming that, if an instruction in the trigger sequence is obfuscated, the trigger is no longer recognized by the HT. This assumption is clearly optimistic since, as previously stated, short dummy instruction can sometimes fail in doing so. Regardless this simplified model can still be used to estimate optimal obfuscation efficiency. With this assumption the probability of breaking a trigger sequence can be computed as:

$$P(\tau = 1) = 1 - (1 - f)^l$$

where τ is a boolean random variable equal to 1 if the trigger is broken, f is the substitution frequency and l is the length of the trigger. Substitution frequency can be controlled by the obfuscator designer and it's a value between 0 (no instruction substituted) and 1 (all the instructions obfuscated). Trigger length depends on the HT designed by the adversary and, even if its unknown to the designer, a few assumptions can be made on its value.

Trigger length its lower bounded by its rarity: the attacker has to design a trigger that is rare enough not to be triggered by test patterns or accidentally by another program. While activating the trigger doesn't guarantee that the Trojan is detected (since its malicious payload its unknown to the designer), is in the best interest of the attacker making this eventuality as unlikely as possible. Since there are around 100 instructions in the OR1200 instruction set a trigger sequence with length 3 offers 10^6 possible combinations, enough to ensure a sufficient rarity. The upper bound trigger length is complexity: in the proposed threat model HT trigger behaves like a finite state machine, where each instruction in the trigger sequence is a state. More instructions in the trigger means more states in the FSM, thus an increased area footprint of the HT. The adversary must keep the HT design as simple as possible in order for the Trojan to lay undetected in the processor. In picture 4.3 can be clearly seen that the probability of breaking a trigger with the proposed obfuscations increases with trigger length and substitution frequency. $P(\tau = 0)$ is in essence the *Survival Rate* of trigger sequences before and after obfuscation.

A global trend for obfuscation that takes into account multiple trigger lengths concurrently is difficult to estimate, since it requires taking into account the probability of a given trigger length to be used.

4.4.2 Execution overhead

Since substitutions are longer than the instruction that they obfuscate, an obfuscated program requires more instructions to complete than the original one. For clarity, two metrics will be used to model the execution overhead: *executed instructions* and *run-time*. Executed instructions are the number of machine instructions executed during a program, while run-time is the effective time taken to complete the execution in clock cycles. In both cases a simple model can be used to estimate the effects of obfuscation:

$$E = (1 - f) + f K_e$$

$$T = (1 - f) + f K_t$$

Both executed instructions (E) and run-time (T) are expected to increase linearly with f by a factor K_e and K_t respectively. Estimating this factors is, unfortunately, a challenging task. In the case of instruction executed, one could assume that the factor is equal to the average substitution length. This would be true if each instruction in the instruction set has identical probability of appearing in the code. Both run-time and instruction count are highly dependent on the program instruction profile: if a program is composed by instructions with a high substitution length the increase in both values would higher. Moreover, as it will be extensively explained in chapter 6, the proposed obfuscator design is capable of dispatching substitutions instructions during instruction cache misses, thus time overhead is expected to be strongly dependent on

the executed program: programs with poor caching performance are less affected than ones with good cache performance.

4.5 Substitution Table example

A substitution table for the ORBIS32 instruction set is presented in appendix B. This table represents only a fraction of all the possible substitutions available for each instruction, and it's used to showcase an example of what a good substitution set should look like. The table has been designed following these constraints:

- The reference instruction should never appear in a true substitution
- Prioritize true substitution, restricting the use of dummy only when necessary
- Prioritize shorter true substitutions when available
- Use only two instruction dummy with padding after the reference

The last constraint is arbitrary, and is only imposed to limit the range of true substitution possible. The following table contains 62 substitution of the 99 ORBIS32 instructions, with 67% of true substitutions. The average substitution length is of 2.31 instructions.

Chapter 5

Obfuscator design

5.1 Introduction

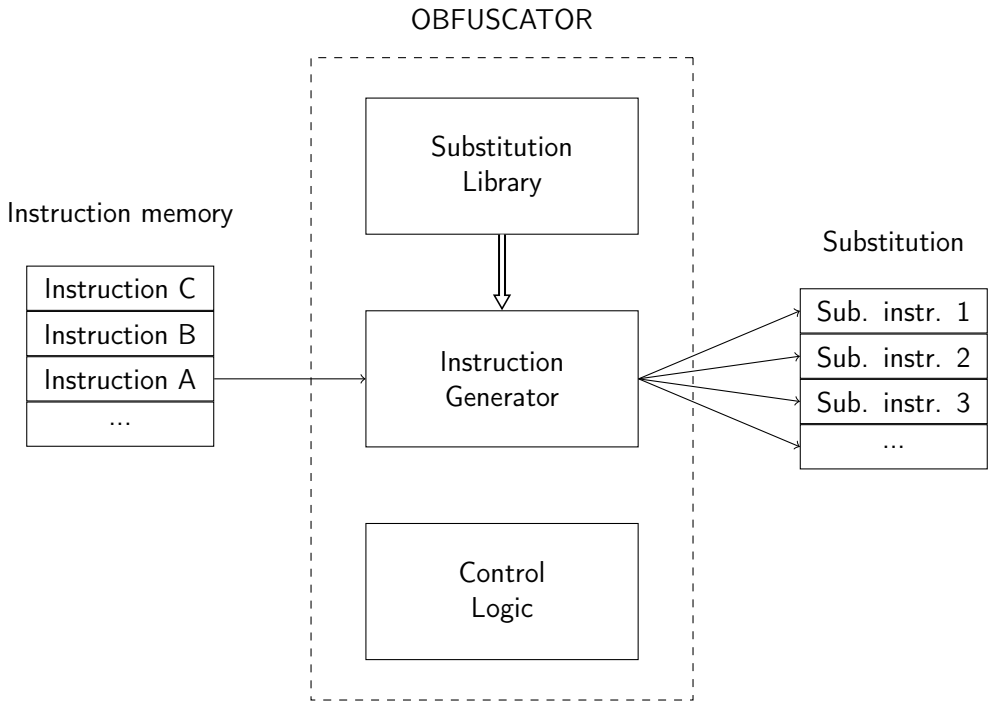

Figure 5.1: Simplified obfuscator model

In this thesis work, it's defined *Obfuscator* the unit in charge of performing on-chip code obfuscation, as well as the set of modification that allows its insertion in an existing processor design. Regardless of the actual implementation, the obfuscator must perform the following tasks:

- Retrieve instructions from memory;

- Decode the rules stored in the library to create a substitution;
- Dispatch the substitution to the decode unit.

A simplified model of the obfuscator is presented in figure 5.1: this doesn't closely resemble the actual implementation, but serves the purpose of better defining the key components of a generic obfuscator design as described in the proposed solution. The unit can be roughly modeled as the combination of three separate elements: the *Substitution Library*, the *Instruction Generator* and the *Dispatch Logic*.

5.1.1 Substitution library

The substitution library is basically a simple memory containing all the substitution for each instruction. The memory is addressed using the reference instruction itself (to select the correct set of substitutions), while the control logic is in charge of choosing a specific substitution according to some difficult to predict algorithm.

5.1.2 Instruction generator

In order for the obfuscator to works, the rules stored in the substitution library must be converted into actual executable machine instruction by the instruction generator. This unit produces the substitutions required to obfuscate the code by combining the reference instruction with the information provided by the library.

5.1.3 Control logic

The control logic it's the core of the obfuscator. It is in charge of controlling the fetching of new reference instructions, the dispatch of substitutions and manage events such as branches and exceptions. This element is also in charge of choosing which instruction to obfuscate, and which instruction to dispatch as is.

5.2 Requirements

Compatibility The obfuscator must not alter the behavior of the processor in any way. This can be summarized by stating that a program produces the exact same results on both the modified and unmodified processors.

Reduced overhead Inserting the obfuscator in an already existing processor design, implies that additional elements have to be added to the system architecture. This immediately raises concerns about the area and delay overhead on the design introduced by the obfuscator. In order to be a suitable solution the overhead must be minimized to a reasonable amount.

Re-usability Since this is a proof-of-concept implementation, the design shouldn't be exclusive to the OR1200 CPU chosen for this project, but should provide a generic concept that is reusable for multiple processors and architectures.

Limited redesign While less critical, it's still favorable to minimize the amount of redesign required to accommodate the obfuscator, mainly to simplify the implementation and produce a cleaner design.

5.3 Insertion points

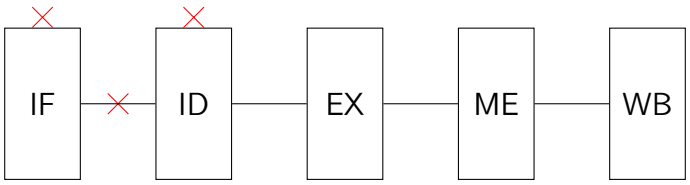

Figure 5.2: Possible insertion points for the obfuscator

In a RISC processor, such as the OR1200 chosen for this project, processor pipeline is formed by 5 separate stages:

1. Instruction fetch (IF)
2. Instruction decode (ID)
3. Execute (EX)
4. Memory (ME)
5. Write-back (WB)

This configuration is known as the *classic RISC pipeline*. To obfuscate an instruction before execution, obfuscation must take place between the IF and ID stage. In figure 5.2 are presented three possible insertion points for the obfuscator. In the following an analysis of each is presented, describing the advantages and disadvantages offered by each choice.

Within IF Inserting the obfuscator within an existing stage has the advantage of requiring limited alterations to the pipeline: with the only exception of some alteration to the control logic, the presence of the obfuscator could most likely be entirely contained in this stage. Moreover, inserting the obfuscator at this point of the pipeline means that all the stages only receive obfuscated instruction, potentially reducing the

attack area. However, inserting the obfuscator in the IF stage means that it has to be completely redesigned to accommodate the new functions. Also, the delay of the stage is incremented by the delay of the obfuscator: this will most likely produce worst performance than having an independent obfuscate stage.

Within ID Similarly to the previous approach, this method shares all the same advantages and disadvantages, but with a further improvement: if the obfuscator is inserted in the ID stage, the instruction generator doesn't have to generate actual instructions, but only the related control signal to the EX stage. This could greatly simplify the "instruction generation" logic and thus reduce the area overhead.

Between IF and ID Inserting the obfuscator between the IF and ID as a separate stage is the best choice for multiple reasons. First of all a pipelined approach allows to reduce the impact of the delay introduced by the obfuscator on the processor throughput. Moreover, using a separate stage, allows the ID and IF to operate independently from the obfuscator: this is especially useful since it allows to dispatch a substitution while the IF stage is stalling, saving clock cycles during instruction cache misses. The only penalty of this approach is that the original pipeline of the OR1200 has to be extensively modified to accommodate the new stage. Regardless, this approach has been chosen for the obfuscator implementation.

5.4 Chosen solution

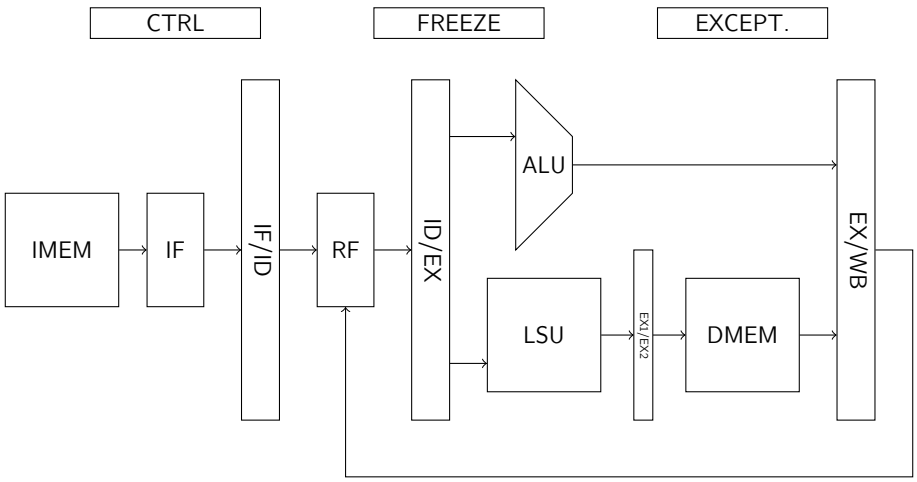

Figure 5.3: OR1200 unmodified pipeline

To implement the on-chip obfuscator a new stage called *Instruction Obfuscate* (IOB) is added to the canonical RISC pipeline, inserted between the IF and ID stages. The IOB

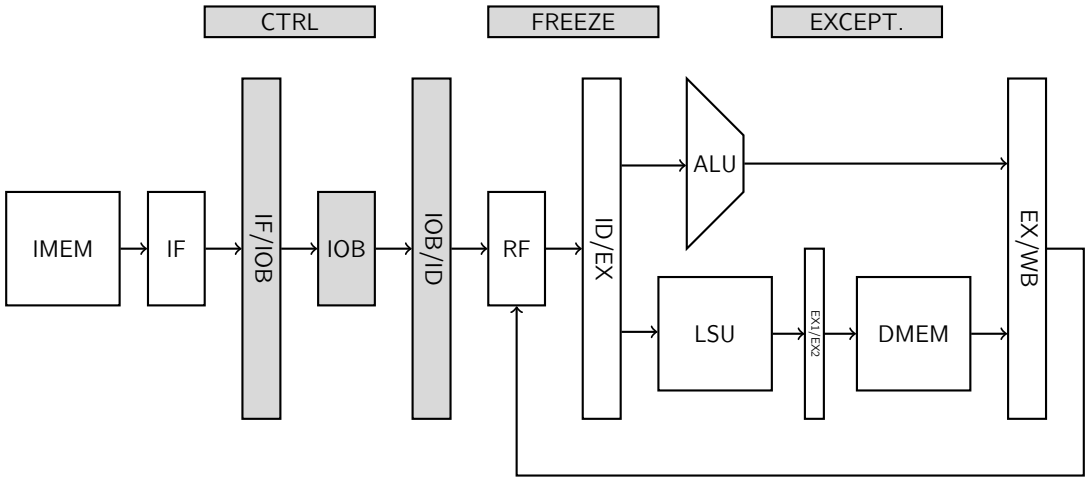

Figure 5.4: OR1200 modified pipeline.

stage mimics ID on its input and IF on its output, to avoid modifying the two adjacent stages in an existing CPU design, allowing these stages to be "unaware" of the presence of the obfuscator. Clearly, the control unit must be modified to handle the new stage and its relationship with the other pipeline stages. Figure 5.3 shows a graphic representation of the OR1200 processor pipeline, and figure 5.4 the modified one including the IOB stage. In the figure, the grayed modules represent the ones that is necessary modify even in a very simple way to be able to insert the IOB stage.

Generally speaking, the obfuscation process goes as it follows: when an instruction is retrieved by the IF stage, it goes through IOB where is replaced with a substitution that may be composed of one or more instructions. The new instruction sequence is then issued to the ID stage, while the fetch cycle is forced to stall. Instruction dispatch is performed sequentially by the obfuscator using an internal counter. To summarize, the obfuscation cycle is:

1. A new instruction is fetched by the IF and passed to the IOB for obfuscation.

2. The obfuscator logic evaluates if a substitution must be performed, if so, it requests an IF stall.

3. A substitution is selected from the substitution library, and the reference instruction is processed to create the code.

4. The substitution code is dispatched to the decode stage and IF stall is dismissed.

Figure 5.5 shows a simple diagram depicting the behavior of the IOB stage.

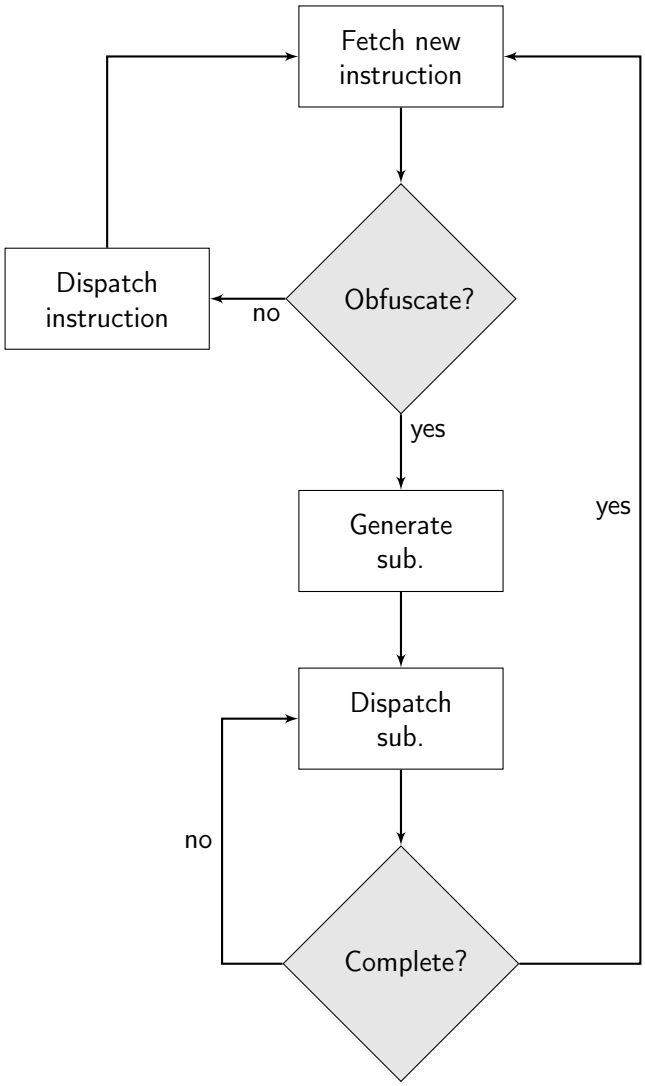

Figure 5.5: Obfuscation logic flow

5.5 Substitution library

Before implementing the proposed IOB stage in the OR1200 pipeline, a suitable way to store and generate substitution instructions must be defined. According to the proposed solution, the obfuscator must be able to:

1. Associate any instruction to a substitution in library
2. Generate the substitution code from the reference instruction
3. Dispatch the code sequentially

The chosen approach is to store substitutions in a set of *Look-Up Tables* (LUTs), each one containing a substitution for each operation in the instruction set: one LUT allows one substitution for each instruction in the instruction set, and thus multiple tables are needed to provide multiple obfuscation alternatives. Each line of the LUT contains a *Substitution Word*, i.e. the informations required to generate a substitution instruction (not an entire substitution).

This approach is interesting for its simplicity and scalability, but raises concerns on the area impact of the obfuscator. The OR1200 has around 100 instructions, and the average substitution length is of 3 instructions. This implies that in order to create a complete substitution set, a LUT must have at least 300 lines. However, the obfuscator must provide more than one substitution for each instruction, in order to perform an efficient obfuscation, so if five alternatives for each instruction are required, five 300 lines LUT have to be implemented in the design! Moreover, each line of the LUT must contain all the informations required to generate new instruction from the reference, so it must contain the substitution opcodes, custom immediate and source/destination register addresses. This implies an amount of information similar to a regular instruction, thus width can be estimated to be 32bit: the substitution library design must somehow try to reduce LUT size.

To design such system two questions must be answered: how to encode substitution words and how to address them.

5.5.1 Addressing

Since substitutions have variable length in terms of number of instructions, LUT addressing must be performed using a pointer table that associates a given reference instruction with the line of the LUT containing the first instruction of its substitution. The entire substitution can be obtained by incrementing this pointer, to retrieve each subsequent substitution word, up to the last one of the sequence, that is marked by the encoding as such. The pointer table is instead addressed by associating to each instruction a unique index. This approach allows to share a substitution, or even a subset of its code, between different instructions, thus reducing the LUT length. The unique index of each ORBIS32 instruction is reported in table A.1.

5.5.2 Encoding

As previously stated, in order to encode substitutions, the LUT must contain the rules required to convert the reference instruction into the substitution code: each line of the LUT corresponds to a single instruction of a substitution. To better understand the requirements of the encoding, here's an example:

Listing 5.1: Example of complex substitution

```
1    # Reference: l.rori rD,rA,L
```

```
2       l.xor  rD,rA,r0
3       l.xori  r0,r0,L
4       l.xori  r0,r0,31
5       l.addi  r0,r0,33
6       l.sll  r0,rD,r0
7       l.srli  rD,rD,L
8       l.or  rD,rD,r0
9       l.xor  r0,r0,r0
```

This substitution, while being a particularly complex one, highlights all the features necessary for the LUT encoding, in order to implement the proposed obfuscation technique. First of all, each instruction of the substitution is different from the reference, and thus the encoding must specify the type of instruction that has to be produced, providing operation code and additional function fields (if the operation has any). This varies greatly depending on the type of instruction that has to be produced. Moreover, the encoding must also describe how to set the operands of the instruction. A peculiarity of substitutions, with respect to regular instructions, is that operands are not predefined: `rD`, `rA`, and `rB` are placeholders for the actual values used by the reference instruction; their values vary according to the actual instruction that is substituted. The same thing happens with immediate operands. The substitution instruction operands are set to either the values of the ones of the reference instruction or custom values encoded in the LUT, depending on the substitution implementation. In the case of register operands, their value could be either `r0`, `rD`, `rA`, `rB`. In the case of immediate operands, their value is either the one in the reference (if any) or a custom one specified in the substitution word. Finally, due to the addressing technique proposed, the substitution word must also mark the last instruction of a substitution, to signal the obfuscator when the substitution has ended.

Unfortunately, not all the instructions of the OR1200 have the same operands, nor the same encoding: this means that the substitution word format must change for each type of instruction, since different operations require different informations to be produced. By studying the instruction set it's possible to classify the OR1K instructions format into 7 separate types:

Type A				
31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
0x38	D	A	B	ALU op.

Type I			
31 - 26	25 - 21	20 - 16	15 - 0
Op.code	D	A	I

Type M				
31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
Op.code	I	A	B	I

Type F				
31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
Op.code	Flag op.	A	B	Res.

Type FI			
31 - 26	25 - 21	20 - 16	15 - 0
Op.code	Flag op.	A	I

Type B	
31 - 26	25 - 0
Op.code	N

Type S	
31 - ?	? - 0
Op.code	N

The type of each ORBIS32 instruction is reported in table A.1.

To summarize the substitution encoding must:

- Define the type of operation
- Handle different operands fields and instruction encodings
- Set the operand to either custom values or the ones in the reference
- Mark the last instruction of the substitution

The proposed encoding is able to achieve all the required features by implementing different formats for each type of instruction (i.e., Type-A, Type-I, etc.). In the substitution word are specified the format of the instruction that it encodes, a command field, whose format is unique for that given type and a stop bit, to signal if the instruction is the last of the substitution. Within the command are specified how to set all the fields of that instruction type, to produce the correct substitution. When the immediate field is set to a custom value in type-M, type-I, and type-FI the value is stored in an additional LUT line after the current one. When this happens, the obfuscator addressing logic has to skip the next line to move to the next substitution word. This allows to limit the width of the LUT to only 16-bit.

The proposed substitution word encoding is the following:

15 - 12	11 - 1	0
Substitution type	Command	Stop

Substitution Type type of instruction that must be generated by the substitution

N-type null substitution, replicates exactly the reference instruction.

A-type arithmetic instruction (opcode 0x38)

I-type immediate-like instruction (operand B not used)

M-type memory-like instruction (operand D not used)

F-type comparison (flag) instruction

FI-type immediate comparison (flag) instruction

Command contains information on how to modify the original instruction to create the substitution; its format varies according to the substitution type.

Stop field is 1 if the encoded substitution is the last one of the sequence, and itâĂŹs used to trigger the fetch of a new instruction.

In the following section is presented an overview of the encoding of the command field for each instruction type.

Type-A command

11 - 8	7 - 4	3	2 -1	0
Op.code 1	Op.code 2	D	A	B

Opcode 1 First auxiliary op. code of the arithmetic instruction

Opcode 2 First auxiliary op. code of the arithmetic instruction

D Destination register

0 Reference instruction D register

1 Force D register to r0

A Operand A register

00 Reference instruction A register

01 Reference instruction B register

10 Reference instruction D register

11 Force A register to r0

B Operand B register

0 Reference instruction B register

1 Force B register to r0

Type-I command

11 - 6	5 - 4	3	2 -1	0
Op.code 0	I	D	A	R

Opcode 0 Op.code of the instruction

I Operand I register

- **00** Original immediate
- **01** Zero immediate
- **10** Custom immediate
- **11** Reserved

D Destination register

- **0** Reference instruction D register
- **1** Force D register to r0

A Operand A register

- **00** Reference instruction A register
- **01** Reference instruction B register
- **10** Reference instruction D register
- **11** Force A register to r0

Type-M command

11 - 6	5 - 4	3	2 -1	0
Op.code 0	I	R	A	B

Opcode 0 Op.code of the instruction

I Operand I register

- **00** Original immediate
- **01** Zero immediate
- **10** Custom immediate
- **11** Reserved

A Operand A register

- **00** Reference instruction A register

01 Reference instruction B register

10 Reference instruction D register

11 Force A register to r0

B Operand B register

0 Reference instruction B register

1 Force B register to r0

Type-F command

11 - 7	6 - 5	4 - 3	2 -1	0
Flag op.	R	B	A	R

Flag op. Flag operation code

A Operand A register

00 Reference instruction A register

01 Reference instruction B register

10 Reserved

11 Force A register to r0

B Operand B register

00 Reference instruction B register

01 Reference instruction A register

10 Reserved

11 Force B register to r0

Type-FI command

11 - 7	6	5 - 4	3	2 -1	0
Flag op.	R	I	R	A	R

Flag op. Flag operation code

A Operand A register

00 Reference instruction A register

01 Reference instruction B register

10 Reserved

11 Force A register to r0

I Operand I register

00 Original immediate

01 Zero immediate

10 Custom immediate

11 Reserved

5.5.3 Library example

To conclude the substitution library topic, in this section is presented a simple example to better understand how addressing and instruction encoding works.

Let's imagine a substitution library containing only two substitutions, one for `l.addi`, the other for `l.exthz`:

Listing 5.2: Substitution for `l.addi`

```
1  # Reference: l.addi rD,rA,I
2  l.add r0,r0,r0
3  l.addic rD,rA,I
```

Listing 5.3: Substitution for `l.exthz`

```
1  # Reference: l.exthz rD,rA
2  l.andi rD,rA,65535
```

This library, clearly, provides only one substitution for each one the instructions, thus is composed by one LUT, and addressed by a single pointer table. Since the substitution for `l.addi` is composed by 2 instructions, and `l.exthz` by one with a custom immediate, the LUT will be composed by 4 lines; a fifth line, containing a *Null-Type* substitution word has to be added to address all the instruction that don't have a substitution. Note that since a pointer table is used to address the content, substitutions can be stored in any order. The content of the LUT is thus the following:

LUT Address	Reference	Content
0	`l.addi`	`l.add r0,r0,r0`
1		`l.addic rD,rA,I`
2	`l.exthz`	`l.andi rD,rA, Custom`
3		`65556`
4	?	Null-Type word

The pointer table is basically a boolean function whose input the unique index associated to the reference operation, and the output is the address of the LUT storing the first instruction of the substitution. The unique index is 7-bit wide while the address can vary according to the size of the LUT addressed: 3-bits are required in this particular case. The resulting pointer table is the following:

Input	Output
...	100
0011011	000
...	100
0111110	011
...	100

All the indexes without substitution point to address 4, the Null-Type substitution, index 27 (`l.addi`) to address 0 and index 62 (`l.exthz`) to address 2.

The only thing left is to actually generate the encoded content of the LUT: the first line contains `l.add r0,r0,r0`, a type-A instruction. As such, the type field is set to '001' and the stop bit is set to 0, since this is not the final instruction for the substitution. The command for a type-A instruction is composed by 5 fields: *AOP1,AOP2,D,A* and *B*. AOP1 and 2 are the function fields of the operation (as specified by the OR1K manual), and for `l.add` are respectively '0000' and '0000'. The destination register is `r0` thus D must be set to 1; for the same reason A and B are set to '11' and '1'. Similarly, the next line contains a Type-I instruction, *l.addic*, so the type field is '010'. Since this is the last instruction of the substitution for `l.addi` the stop bit is set to '1'.

This procedure can be repeated for all the remaining substitution instructions; the resulting LUT content is:

LUT Address	Reference	Sub. word
0	`l.addi`	0010000000011110
1		0101010000000001
2	`l.exthz`	0101010011000001
3		1111111111111111
4	?	0000000000000001

Chapter 6

Obfuscator implementation

6.1 Introduction

After all the characteristics of the obfuscator were laid down, a proof-of-concept implementation of the modified CPU was developed using Verilog HDL. Using as a starting point the default OR1200 configuration, the IOB stage was designed and added between IF and ID. This however took more time than expected due to the lack of detailed documentation on the OR1200 design. The development required a preliminary stage of reverse engineering of some key components, by carefully studying the signals produced during the execution of some ad-hoc programs, to generate specific events (e.g., jumps, stall, exceptions, etc.) in a controlled environment. Regardless, inserting the new stage in the design still required an extensive amount of tuning by trial-and-error to achieve a functional implementation. For this reason, the modified CPU has been extensively tested over the course of multiple weeks using many benchmark programs, to identify errors and misbehaviors, by comparing the execution of the modified processor with the original reference design.

Unfortunately, this time-consuming approach meant that some planned features had to be dropped due to time constraints: without detailed documentation, modifying a previously unknown element of the system can take an indefinite amount of time to deeply understand its function, let alone modify it. The final system is fully functional, in the terms presented in the previous chapters, and thus can fetch, generate and dispatch substitutions, but lacks exception management. While this feature was technically implemented, it never worked with absolute reliability.

This chapter presents a detailed analysis of the implementation of the modified OR1200 CPU, focusing primarily on the elements most affected by the obfuscator, namely the instruction fetch and decode stages, as well as the control logic around them. The last sections will instead deal with missing exception management, the issues that brought to the design as well as possible solutions to it.

6.2 Original pipeline

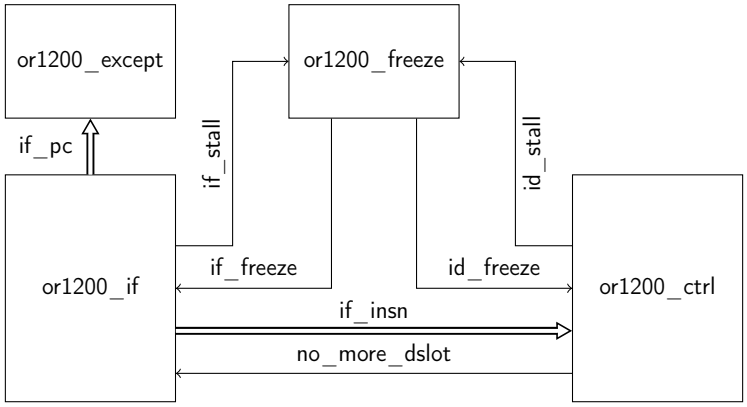

Figure 6.1: Detail of the unmodified IF and ID stage

To better understand the modified CPU, a little of background on the unmodified version is necessary. In figure 6.1 presents a detail of the wiring between the first two stages of the pipeline (i.e., IF and ID). In the OR1200, there's no single control unit, and its role is taken by multiple independent elements each implementing a specific aspect of the control logic. These elements are: *or1200_except*, that manages exceptions, *or1200_freeze*, responsible from handling stalls, and *or1200_ctrl*, that handles some aspects of flushing and branching, and also acts as the ID stage.

The role of the IF stage is performed by the module *or1200_if*: this module sends the fetched instruction to or1200_ctrl trough the *if_insn* bus to be decoded or, if no instruction is available, raises the signal *if_stall* to notify the freeze logic of the event. In the case of a fetch stall a special nop instruction, called *void*, is passed to the ID stage: void acts like the default state of instruction registers after either flush or reset. Similarly, an ID stall is signaled using the *id_stall*. The signal *no_more_dslot* is used to notify the IF stage that a branch has been taken and that the instruction currently decoded is the branch delay slot: this triggers an immediate flush of the instruction in the IF stage.

The program counter corresponding to the fetch instruction is passed to the exception management logic via the *if_pc* bus: the unit keeps track of the PC value corresponding to the instruction in each stage of the pipeline, to correctly determine the last executed instruction to return from an exception. Each stage of the pipeline has an independent flush signal wired to this unit (not reported in figure 6.1): when an exception occurs, the stages before the one where it was generated are flushed, allowing precise exception handling.

6.3 Modified pipeline

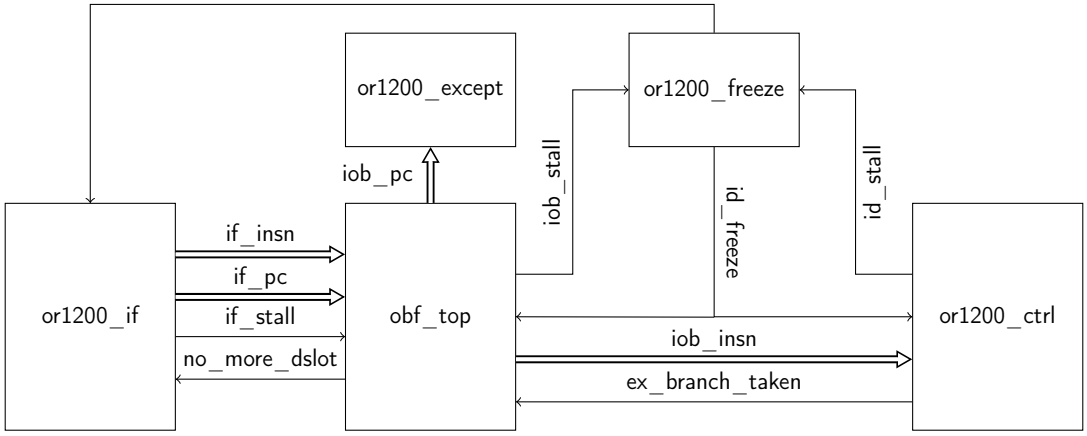

Figure 6.2: Detail of the modified IF and ID stage

As explained in the previous chapter, the obfuscator, the *obf_top* module in figure 6.2, behaves like a new stage inserted between IF and ID. For these reasons, it basically acts like ID on its inputs and IF on its outputs, and thus neither of these stages have to be modified to insert the new unit; only or1200_except and or1200_freeze require some sort of alteration to insert the new stage. Control signals, connected directly between IF and ID, are now rewired trough the obfuscator, that controls their value: the IOB logic has thus an active role in integrating the obfuscator with the pipeline. This allows to keep the original design of most modules unmodified, at the expense of a slightly more complex obfuscator logic.

In the modified CPU, the output signals coming from the IF stage are passed trough the obfuscator, that controls their value to manage instruction execution. Thus, a number of signals produced by the or1200_if, in the modified processor, are produced by the new stage. These signals are if_insn, if_stall and if_pc, or in simpler words the fetched instruction, the fetch stall signal and the program-counter value; these are mirrored at the output of the IOB stage, and wired to their original destination. The obfuscator takes the instruction on the if_insn bus and uses it to create the substitutions that are then passed to ID in *iob_insn* bus. *iob_stall* works similarly to its IF counterpart: if the obfuscator has no instruction to dispatch the signal is asserted and the event is handled by the freeze unit. This signal can either be controlled by the substitution dispatch logic, while a substitution is performed, or mirror the if_stall signal, when the IOB stage is idling. Similarly, also the program counter value, *if_pc*, is controlled by the obfuscator with the *iob_pc* signal: while a substitution is dispatched the PC value of the reference operation is used for every instruction.

A similar approach, but in the opposite direction, is done with the no_more_dslot signal: in the original pipeline this ID signal, used to notify that the delay slot is in

the ID stage when a branch is taken, is wired directly to the IF stage. In the modified processor, the original signal from ID is no longer connected to anything and it's logic is reimplemented within the obfuscator from scratch, to take into account the new module.

The obfuscator can trigger an independent IF freeze trough the *if_freeze_req* signal (not reported in figure 6.2): this is used to stop the fetch cycle while a substitution is being dispatched. During obfuscation, the IF stage is allowed to fetch a new instruction and, as soon as it's retrieved, a freeze request is issued. This allows optimizing execution time by dispatching instructions while new ones are fetched from memory. Other than that, the freeze logic can be used as is; the IOB stage has no independent freeze control, and reuses the id_freeze signal.

6.4 Obfuscator architecture

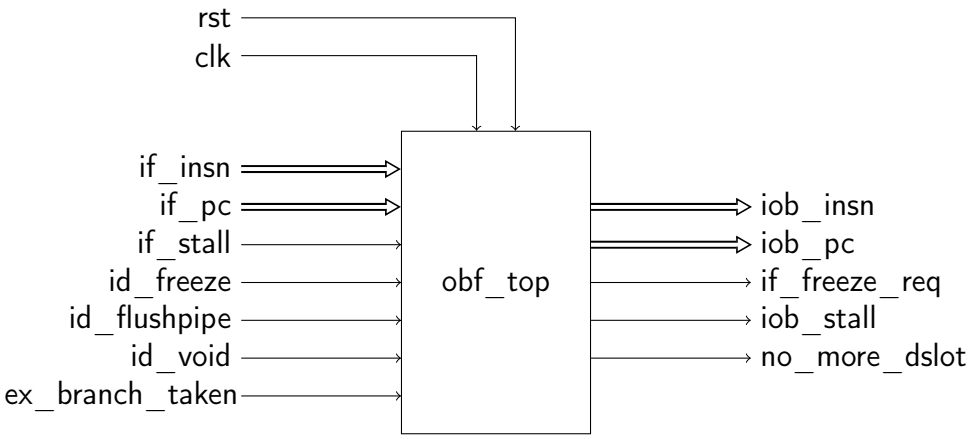

Figure 6.3: Top level view of the obfuscator

A top-level view of the obf_top module is shown in figure 6.3. The input/output signals of the obfuscator are the following:

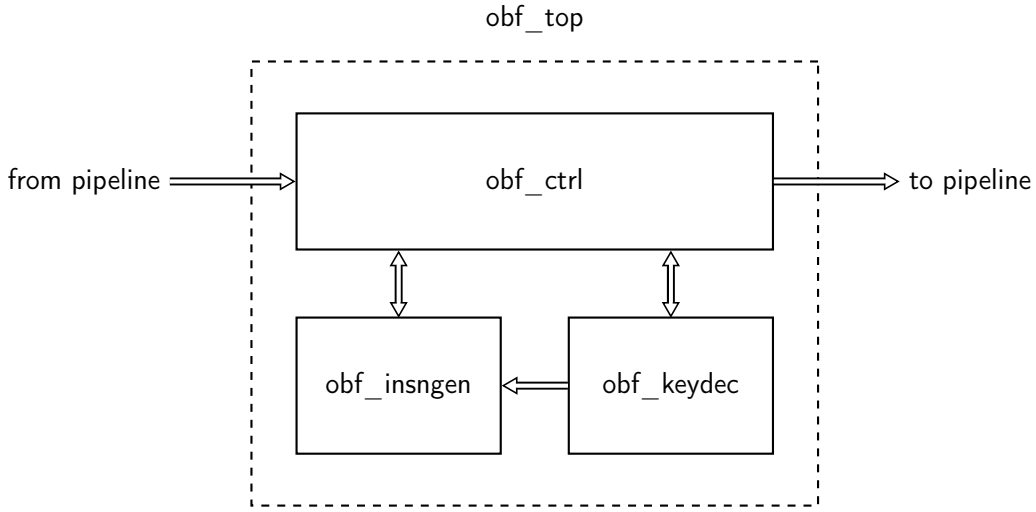

Figure 6.4: Obfuscator architecture

Signal	Type	Width	Description
if_insn	Input	32	Fetched instruction bus
if_pc	Input	32	Fetched instruction program counter
if_stall	Input	1	IF stall signal
id_freeze	Input	1	ID freeze signal
id_flushpipe	Input	1	ID flush signals
id_void	Input	1	Asserted when a void instruction is in ID
ex_branch_taken	Input	1	Asserted when a branch is taken
iob_insn	Output	32	Obfuscated instruction bus
iob_pc	Output	32	Obfuscated instruction program counter
iob_stall	Output	1	IOB stall signal
if_freeze_req	Output	1	Triggers IF freeze when asserted
no_more_dslot	Output	1	Signals IF the delay slot decode

The first three signals, namely if_insn, if_pc and if_stall, are IF outputs that in the unmodified processor were directly connected to the ID stage, that are now wired through the obfuscator and mirrored on its output as *iob_insn*, *iob_pc* and *iob_stall*. The remaining inputs are sense signals used by the obfuscator logic to control the dispatch of substitution. Note that no_more_dslot is a completely different signal with respect to the one present in the original processor, where it was generated by the ID stage: the original signal, in this implementation is no longer used.

Internally the obfuscator is composed by three separate elements, as depicted in picture 6.13: *obf_ctrl*, the control unit of the obfuscator, *obf_insngen*, the unit tasked with generating substitution, and *obf_keydec*, that implements the algorithm used to select substitutions from a unique configuration called "Key". In the following sections

a detailed description of each one of these elements is presented.

6.4.1 Control unit

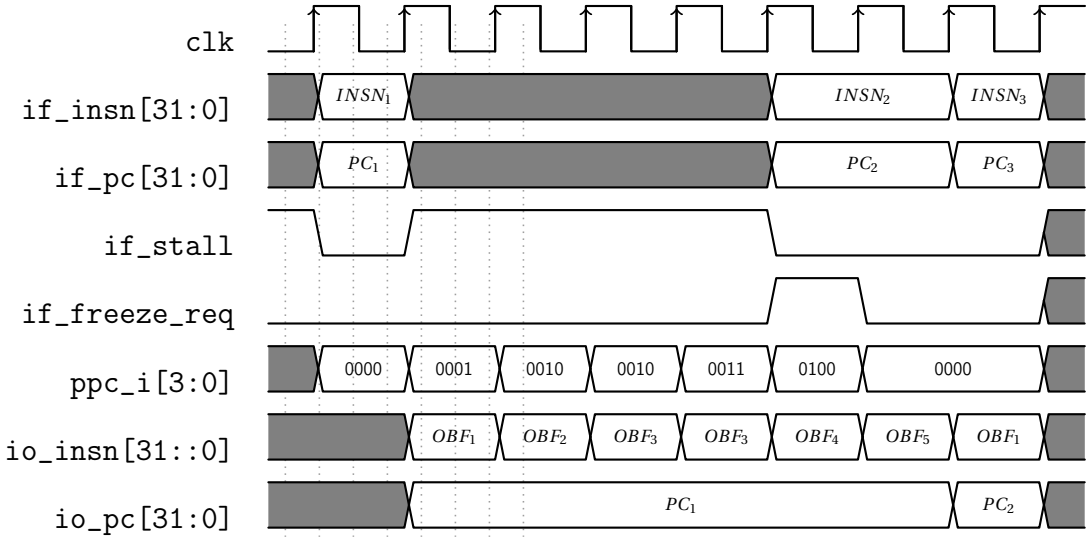

Figure 6.5: Instruction fetch and substitution dispatch timing

The control unit is responsible for handling the instruction fetch/substitution dispatch cycle, as well as ensuring the correct behavior of the obfuscator during stalls, branches and exceptions. This unit is implemented as multiple behavioral logic elements, namely:

- Pseudo-program counter
- Input latch logic
- Fetch freeze request logic
- Substitution dispatch logic
- Branch management

The core of the control logic is the *Pseudo-program counter* (PPC), a counter used to address the substitution library and incrementally dispatch the instruction, by using its value as offset for the pointer table. The entire logic of the control unit is built around this counter.

Instruction fetch/substitution dispatch cycle is presented in picture 6.5. When a new instruction is fetched by the IF stage, the if_stall signal is set to 0 and the new instruction is available on the if_insn bus with its corresponding PC value on if_pc. These

two register values are latched into two registers to store them for later instruction substitutions. If the ID stage is available (id_freeze set to 0) the first instruction of the substitution will increment the PPC value in this clock cycle, otherwise the PPC is not incremented. During the obfuscation, the IF stage is allowed to fetch another instruction to save clock cycles: if a new instruction is available before the substitution dispatch has finished, the obfuscator will trigger an IF freeze trough the if_freeze_req signal, that will be left active until the last instruction of the substitution has been reached. If the ID stage is not frozen the PPC will be incremented each clock cycle until a substitution with the stop field equal to 1 is reached: this triggers the reset of the PPC and a new instruction is parsed by the obfuscator.

Pseudo-program counter

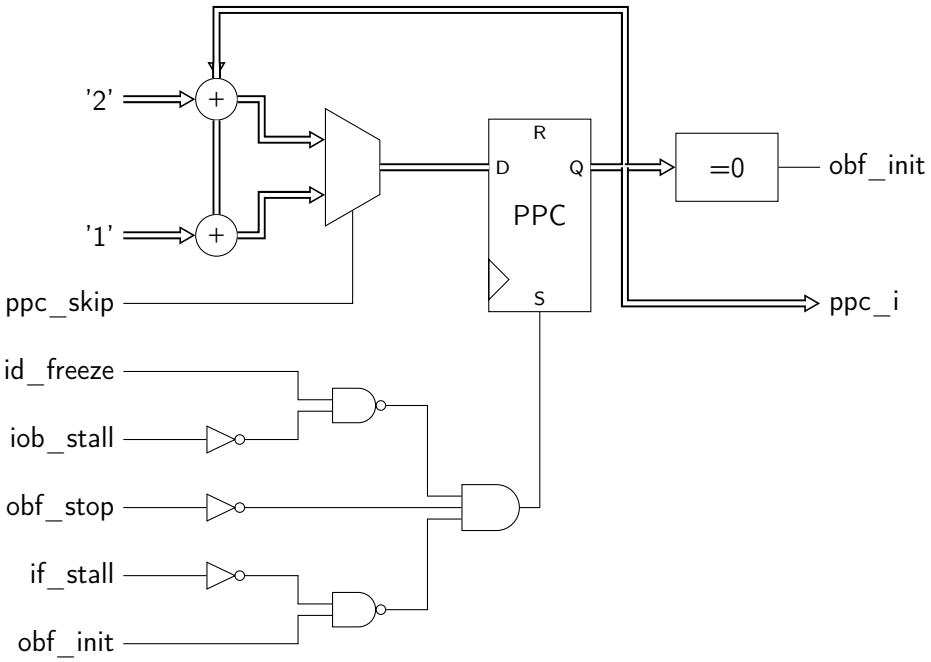

Figure 6.6: RTL implementaion of the pseudo program counter

The Pseudo-program counter has a similar purpose with respect to a regular program counter, in the sense that is used to address instructions: the PPC value is in fact used as offset to the substitution library pointer table, to address all the instructions of a substitution. For this reason the PPC width is very small, only 4-bit in the proposed implementation. Since in the library LUTs, a substitution word can have an optional "custom immediate" in the following line, the PPC can either increment by 1 or 2 units, depending on the value of the *ppc_skip* signal, produced by the instruction generator. The counter value is incremented each clock cycle up to the last instruction

of the substitution, that triggers the PPC reset. Regardless, the PPC count can be frozen for various reasons: if the ID stage is frozen, while waiting for a new instruction from IF or if the branch management requires it.

Input latch logic

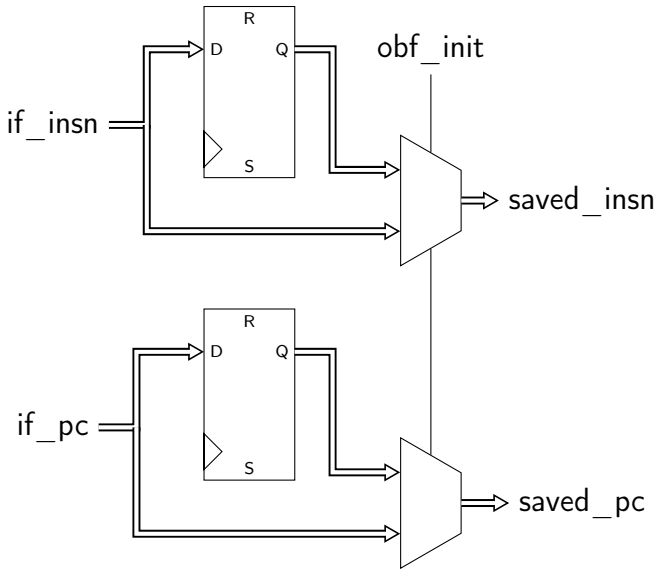

Figure 6.7: RTL implementaion of input registers

During the obfuscation cycle, the IF stage is allowed to fetch a new instruction while a substitution is being dispatched, thus the reference instruction and its PC value (if_insn and if_pc respectively) are no longer present at the output of the instruction fetch stage in later cycles. These values are thus latched into two internal registers after the first clock cycle to allow the IF stage to operate independently (figure 6.7).

The mechanism is quite simple: if the PPC value is zero (obf_init signal is 1), the reference instruction, and its relative PC value are latched at the output of the IF stage. The obfuscator input is muxed to either the IF outputs, if obf_init is 1, or its internal registers, if obf_init is 0. Internal registers are enabled by the obf_init signal, and thus store the values only during the first obfuscation cycle.

Fetch freeze request logic

In the modified processor the Freeze unit is no longer in direct control of the instruction fetch stage, and triggers the if_freeze signal only if if_freeze_req is asserted by the obfuscator. The obfuscator thus is in charge of freezing the IF stage: this stage must be frozen if the ID stage is stalling and while dispatching a substitution. The freeze request logic is described by the following Verilog code:

Listing 6.1: IF freeze request logic

```
1 always @(*)
2 begin
3   if(if_stall) begin
4       // If fetch stage is stalling no action is required
5       if_freeze_req <= 1'b0;
6   end
7   else begin
8     if (obf_rst)
9       // Obfuscator reset
10      if_freeze_req <= real_id_freeze;
11    else
12      // Obfuscator running
13      if_freeze_req <= (!obf_init & !io_no_more_dslot) |
            real_id_freeze;
14    end
15  end
16 end
```

If IF is stalling, the freeze request is disabled, since it would have no effect on the stage. In all other cases the freeze request is directly wired to the *real_id_freeze* signal: this signal is asserted when id_freeze is high and iob_stall is low, or in simpler words, when ID is frozen but not because the obfuscator is stalling.

If the IF stage is not frozen, the if_freeze_request value depends only on the values of obf_init and no_more_dslot. If the latter is asserted the freeze has to be immediately lifted to allow flushing the fetched instruction. The former is instead used by the obfuscation dispatch cycle: if obf_init is set to 1, thus the PPC is equal to 0, the obfuscator has just received an instruction from IF, thus if_freeze_request is set to 0 to fetch a new one in the next cycle. After that, the program counter is incremented and obf_int becomes 0: if the fetch stage is stalling, the freeze request is removed until the new instruction is fetched, if the instruction is immediately available if_freeze_req is set until the next obfuscation cycle.

Substitution dispatch logic

The substitution dispatch logic controls the iob_insn, iob_pc and iob_stall signals to send the obfuscated instruction produced by the instruction generator to the decode stage. The Verilog implementation is the following:

Listing 6.2: Substitution dispatch logic

```
1 always @(posedge clk or `OR1200_RST_EVENT rst)
2   begin
3   if (rst == `OR1200_RST_VALUE | obf_rst) begin
4     // Reset
5     io_insn <= {`OR1200_OR32_NOP, 26'h041_0000};
6     io_pc <= 32'h00000000;
```

```
 7      io_stall <= 1'b0;
 8    end
 9    else begin
10   if (real_id_freeze) begin
11        // Stop
12        io_insn <= io_insn;
13        io_pc <= io_pc;
14        io_stall <= 1'b0;
15      end
16      else if (obf_init & if_stall) begin
17        // Bypass
18        io_insn <= if_insn;
19        io_pc <= if_pc;
20        io_stall <= 1'b1;
21      end
22      else begin
23        // Obfuscate
24        io_insn <= obf_insn;
25        io_pc <= saved_pc;
26        io_stall <= 1'b0;
27      end
28    end
29  end
```

As it can be seen from the code, the dispatch logic can either operate in four states with decreasing priority:

- Reset
- Stop
- Bypass
- Obfuscate

Reset state can be either external, at system reset, or internal, after branches or exceptions: it resets the content of the output registers to their default state. Stop mode is instead activated if the ID stage is stalling, and the output values are kept unmodified. Bypass mode is used while waiting for a new instruction to be fetched by the IF stage (obf_init and if_stall to 1): in this case the signal from if are mirrored directly to the output, with the exception of iob_stall that is always 1. Finally, the last possible state is obfuscate: in this case, iob_insn is the instruction produced by the instruction generator, iob_pc the latched value of the reference program counter and iob_stall is set to 0.

Branch management

In the OR1200 default configuration branching is performed with a branch delay slot: the branch logic ensures that the delay instruction is executed before flushing the pipeline.

This is done trough the use of a special instruction called *void*, a `l.nop` with bit 16 set to 1. This instruction is set on the output of the IF stage during stall and is processed by the pipeline like a regular instruction. If a branch is taken, the branch logic checks if a void currently in either the IF or ID stage, and behaves according to the following table:

IF void	ID void	Behavior
✓	✓	Branch delay slot has not yet been fetched, no action is taken.
✗	✓	Branch delay slot has just been fetched, no action is taken.
✓	✗	Branch delay slot is in ID stage, IF is flushed.
✗	✗	Branch delay slot is in ID, IF stage is flushed.

In the unmodified OR1200 the IF stage flush is triggered by the *no_more_dslot* signal. The modified pipeline mimics this mechanism, with the major difference that obfuscator introduces an instruction of delay. When a branch is taken there are three possible situations: the branch delay slot is being obfuscated, the branch delay slot has already been passed to the ID stage, or the branch delay slot has not yet been fetched. The IO stage can be flushed similarly to the IF stage, but only if the first instruction of a substitution is latched on its output. If the last instruction dispatched to the ID stage had PPC equal to 0 the IO stage can be flushed with the IF stage.

6.4.2 Instruction generator

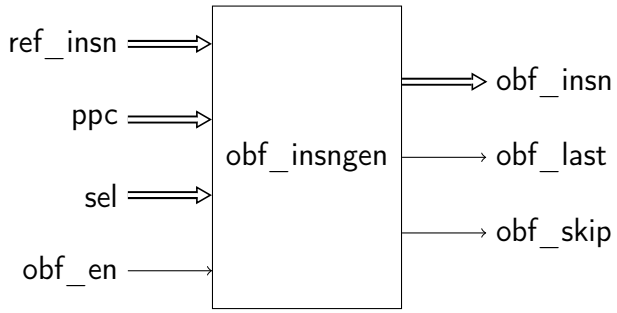

Figure 6.8: Top level view of the instruction generator

The instruction generator is the unit responsible for producing the substitution instructions to be dispatched to the ID stage: this is done by parsing the reference instruction to retrieve its operands, and then use them to build the new instruction according to the rules stored in the substitution library. This block is completely combinational and is entirely controlled by the obfuscator control unit and the key decoder.

Among its inputs are *ref_insn* and *ppc*, respectively the reference instruction and the current PPC value. *sel* and *obf_en*, are controlled by the key decoder and are used to create the unique obfuscation pattern by choosing which instruction to substitute and

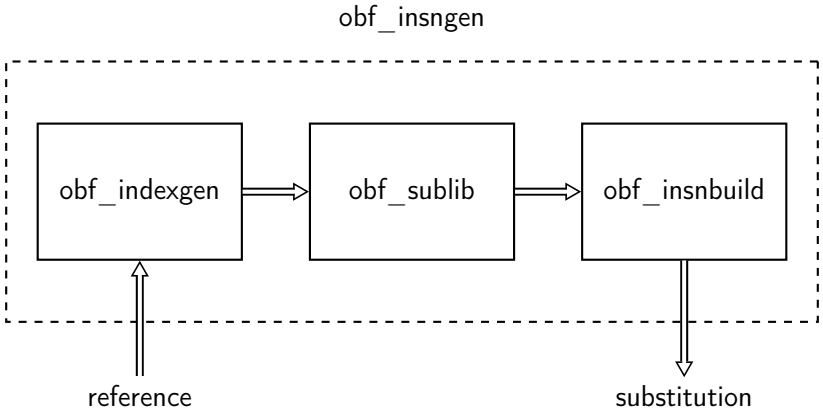

Figure 6.9: Instruction generator architecture

with which substitution. If obf_en is set to zero, the instruction returns the reference instruction as is.

The instruction generator has only three outputs: *obf_insn* is the obfuscated instruction for the current reference, *obf_last* is set to 1 if is the last instruction of a substitution and *obf_skip* is asserted to increment the PPC by 2 units.

Internally this unit if formed by three modules:

- Index generator
- Substitution library
- Build logic

Index generator

The index generator is a combinational block that converts the fetched 32-bit instruction that has to be substituted, into the unique 7-bit index used to address the substitution library. From an implementation point of view, this module was designed using a purely behavioral description, and it amounts to essentially a large set of nested switch statements. This is done because different instructions may have the same operation code, but different functional codes.

Substitution library

The substitution library is implemented exactly as it was described in the previous chapter: it consists in a set of N separate LUTs, each one paired with a pointer table, addressed using the index from the index generator. The value from the pointer table refers to the address of the first instruction of a substitution in its relative LUT; PPC value is used as offset to retrieve the following ones. Such library allows for N possible

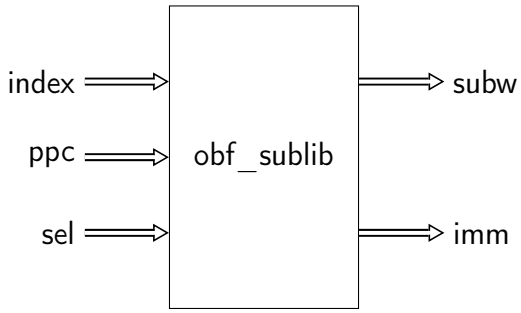

Figure 6.10: Top level view of the substitution library

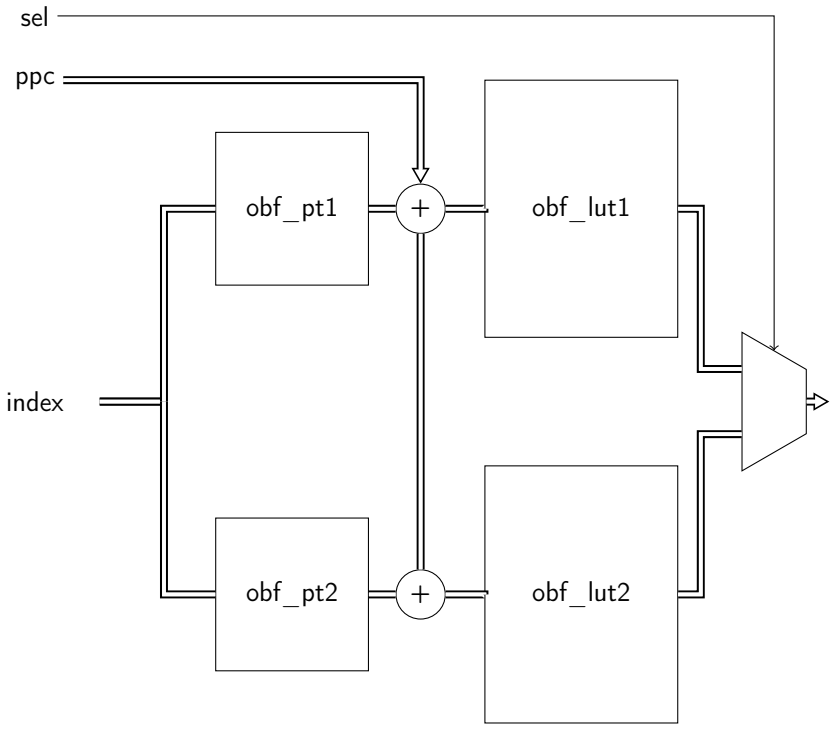

Figure 6.11: RTL implementation of the substitution library

substitutions for each operation of the instruction set: tables are addressed concurrently and their output is multiplexed to select only one substitution (figure 6.11).

A top-level view of the substitution LUT is available in picture 6.10. Buses *index* and *ppc*, are used for addressing. The substitution word is available on the *subw* bus. If the selected substitution word has an immediate field, its value is available on the *imm* bus, in all the other cases imm contains the following line of the LUT.

Build logic

The information from the substitution library has to be converted into actual executable 32-bit instruction: this task is handled by the build logic. First of all the reference instruction type is detected, to correctly retrieve operands. This is particularly critical in the case of type-I and Type-M instructions, where the 16-bit immediate in two completely different positions in the 32-bit.

Listing 6.3: Reference operands parsing logic

```
wire [5:0]  f_in_opc = ref_insn[31:26];
wire [4:0]  f_in_D   = ref_insn[25:21];
wire [4:0]  f_in_A   = ref_insn[20:16];
wire [4:0]  f_in_B   = ref_insn[15:11];
wire [15:0] f_in_I   = (f_in_type == `OBF_INSN_TYPE_I) ?
    ref_insn[15:0] : {ref_insn[25:21], ref_insn[10:0]};
```

Substitution library output is used to build the operands for the substitution instruction: operands are produced independently from the instruction and then combined together to create it. The bus *sw_cmd* contains the substitution word command field, while *sw_type* the instruction type.

Listing 6.4: Operands generation logic

```
wire [5:0] f_out_OPC   = sw_cmd[11:6];
wire [3:0] f_out_AOPC1 = sw_cmd[11:8];
wire [3:0] f_out_AOPC2 = sw_cmd[7:4];
wire [4:0] f_out_FOPC  = sw_cmd[11:7];
wire [4:0] f_out_D     = sw_cmd[3] ? 5'b00000 : f_in_D;

wire [4:0] f_out_A     = sw_cmd[2:1] == 2'b00 ? f_in_A:
                         sw_cmd[2:1] == 2'b01 ? f_in_B:
                         sw_cmd[2:1] == 2'b10 ? f_in_D:
                         5'b00000;

wire [4:0] f_out_B     = (sw_type == `OBF_INSN_TYPE_F ||
                          sw_type == `OBF_INSN_TYPE_FI) ?
                          sw_cmd[4:3] == 2'b00 ? f_in_B:
                          sw_cmd[4:3] == 2'b01 ? f_in_A:
                          5'b00000:
                          sw_cmd[0] ? 5'b00000 : f_in_B;

wire [15:0] f_out_I    = sw_cmd[5] ? lut_out_imm : sw_cmd[4]? 16'd0
    : f_in_I;
```

Depending on the type of substitution instruction required operands are combined to create the correct instruction:

Listing 6.5: Operands concatenation logic

```
always @(*)
```

```
2  begin
3    if(obf_en & !ref_void) begin
4      // Obfuscator enabled
5      case(sw_type)
6        // Type-N
7        `OBF_INSN_TYPE_N: obf_insn = ref_insn;
8        // Type-A
9        `OBF_INSN_TYPE_A: obf_insn = {`OR1200_OR32_ALU,
10       f_out_D, f_out_A, f_out_B, 1'b0, f_out_AOPC1, 2'b00,
             f_out_AOPC2};
11       // Type-I
12       `OBF_INSN_TYPE_I: obf_insn = {f_out_OPC, f_out_D, f_out_A,
             f_out_I};
13       // Type-M
14       `OBF_INSN_TYPE_M: obf_insn = {f_out_OPC, f_out_I[15:11],
             f_out_A, f_out_B, f_out_I[10:0]};
15       // Type-F
16       `OBF_INSN_TYPE_F: obf_insn = {`OR1200_OR32_SFXX, f_out_FOPC,
             f_out_A, f_out_B, 11'd0};
17       // Type-FI
18       `OBF_INSN_TYPE_FI: obf_insn = {`OR1200_OR32_SFXXI,
             f_out_FOPC, f_out_A, f_out_I};
19       // Other
20       default: obf_insn = ref_insn;
21     endcase
22   end
23   else begin
24     // Obfuscator disabled
25     obf_insn = ref_insn;
26   end
27 end
```

6.4.3 Key decoder

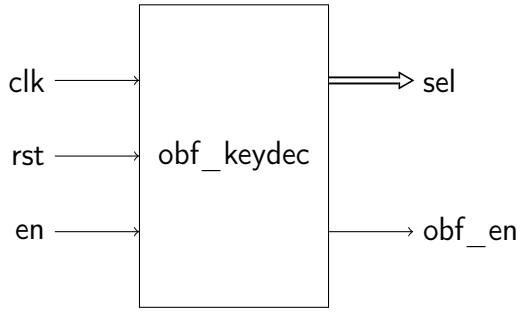

Figure 6.12: Top level view of the key decoder

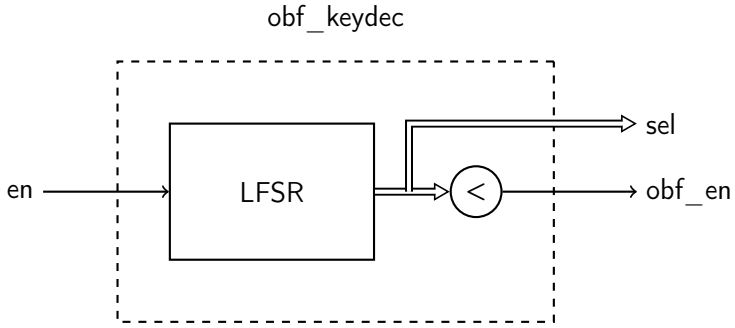

Figure 6.13: Key decoder architecture

The obfuscation pattern is made difficult to predict by using an algorithm configured after manufacturing: this external configuration is referred as "key". The key decoder is responsible of generating all the key-dependent control signals within the obfuscator. Since no specification have been made on the requirements of such entity, for the purpose of this preliminary implementation, the key decoder has been implemented as a mock-up. This module has two main purposes: chose the instructions to obfuscate and choose the substitution to use. Since that, according to the threat model, no assumption can be made on the most effective obfuscation pattern, a random approach has been chosen. Regardless of the lack of specifications about the "key", the CPU user should have at least some sort of control over two parameters of the obfuscation pattern, namely:

- The seed of the random algorithm, to ensure that the pattern uniqueness
- The frequency of substitutions, since it's estimated that obfuscation will produce a run-time overhead

To achieve this, random, or more accurately pseudo-random, number generation is produced by a 32-bit *Linear-feedback shift register* (LFSR). A LFSR is a shift-register whose input is a function of its current content: with a well-chosen feedback function, the sequence produced by the LFSR is a periodical mathematically predictable sequence, that can be considered random in most applications[15]. Externally, the key decoder has only one input (excluding as usual clock and reset signals) call *en*. This signal enables the shift registers and it's wired to the obf_init signal: each obfuscation cycle a new random number is generated. The CPU user can control the obfuscation pattern by changing the seed of the LFSR.

A subset of the LFSR output, is wired to the output signal *sel*, that controls the output mux of the substitution library, allowing a random choice of substitution: the exact width of this control signal depends on the number of LUTs in the library and is thus variable. To achieve fine control over the number of substitution performed per executed instructions, the lower 8-bit of the LFSR was wired to one input of a comparator,

while the other was wired to a constant value. The output of the comparator is directly connected to the *obf_en* output: if the value within the lower byte of the shift register is smaller than the constant, the obfuscation is enabled. This allows to control the substitution frequency through the value of the constant. While this method is rudimentary, it serves the purpose of creating a simple pseudo-random configurable obfuscation pattern.

The slice of 8-bit was chosen to achieve a more than percentual degree of control over the substitution frequency: the value distribution over 1000 iterations of the LFSR is reported in table 6.14, while the substitution frequency obtained at different comparator settings is in table 6.15.

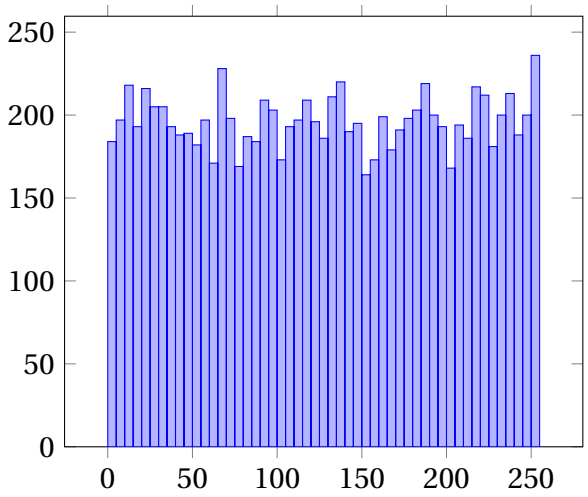

Figure 6.14: LFSR comparator input value distribution

6.5 Failed attempt at exception handling

While not functional, a great design effort was given to the development of the obfuscator exception management. In the unmodified processor, the exception management is performed by the or1200_except module. Within this component is present a sort of "scale model" of the OR1200 pipeline: the exception module keeps track of the program counter in each stage of the pipeline by moving the if_pc value in a register chain, in the same way that instructions are moved trough the pipeline. When an exception occurs, the exception unit stores the PC of the instruction after the one that has caused the exception and flushes all the stages before it. This however can no longer be performed in the modified processor: if an exception happens during a substitution, data is most likely corrupted, and the program is irreparably compromised. The solution to this problem could be two:

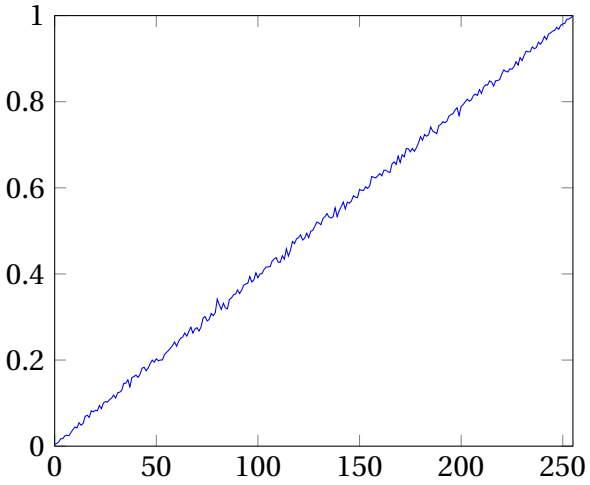

Figure 6.15: Substitution frequency for different comparator values

- If a substitution is suspended by an exception, after execution is resumed, allow the obfuscator to continue from where it left
- Allow the obfuscator to complete the substitution before handling the exception

While both solution are potentially viable, the latter ensures the processor is in the exact same state after the exception as an unmodified one, and its much easier to implement. The design idea was to propagate in the exception module the value of obf_init, similarly to what the module does with the PC value, to keep track of substitutions in pipeline stages. If the instruction in a given stage had obf_init equal to 0, when it was dispatched from the obfuscator, it means that the instruction was either the first instruction of a substitution or a single unobfuscated operation: such instruction, as well as all the others backward in the pipeline, can thus be flushed without breaking any substitution. On the contrary, if an instruction had obf_init equal to 1, it means that forward in the pipeline at least one operation from the same substitution is being executed. This idea however didn't cope well with the selection of the return program counter, resulting in the processor executing the same instruction twice in some rare cases. Unfortunately, the exact cause of this has yet to be found, and focus was shifted to verifying the performance of the obfuscator.

Chapter 7

Experimental results

7.1 Overview

After the design was completed, the system has been subject to a broad set of tests to verify its correct behavior, and estimate its performance. In both cases these were performed by studying the execution of benchmark programs, in an effort to acquire meaningful data in experiments resembling a real-life application/attack scenarios.

The system under test consists in the modified OR1200 in its default configuration, implementing the substitution library described in section 4.5. This library consists in a single substitution table (one substitution for each instruction), a configuration quite different from the proposed obfuscator implementation, that in a real-life application would implement multiple LUTs. This has been chosen for multiple reasons: a single substitution table allows for a more straightforward and predictable substitution pattern, that simplifies the verification process, without compromising the performance evaluation. In fact, multiple substitutions are implemented to prevent attacks targeting the obfuscation pattern, not to improve the obfuscation by itself. Moreover, less LUTs means that fewer substitutions have to be verified and tested.

The benchmark used for these experiments is the Automotive suite of Mibench v1.0, modified to run on the OR1200. All the presented tests were executed using Verilator, a fast verilog simulator, or on a DE10-Lite FPGA by Terasic.

7.1.1 Mibench

Mibench v1.0 is an open-source benchmark suite developed at the University of Michigan [16]. The benchmarks contain six sets of programs, designed to target specific areas of the embedded market. These sets are: Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. The Automotive set has been selected as benchmark for the experiments. It is composed by four programs: basicmath, bitcount, qsort, and susan. *susan* performs a set of

simple mathematical calculations like cubic function solving, integer square root and angle conversions.*bitcount* is a collection of different bit counting algorithms. *qsort* sorts a large string array using the well-known quick sort algorithm. *susan* is an image recognition package used for Magnetic Resonance Images of the brain. It can smooth an image or find corners and edges. All of these programs are executed either with a "small" or a "large" input dataset.

In the proposed tests, these programs have been modified to run in a bare-metal environment without a file system, and output data directly to the testbench. Furthermore, the program bitcount, which uses the internal timer to compute its execution time, has been modified to remove this feature since it generated exceptions and the obfuscator is not able to handle them correctly. Finally, a new configuration of the program basicmath, called "tiny", was created since both the "small" and "large" configuration took too much time to execute.

7.1.2 Verilator

Verilator is a Verilog simulator that converts syntesizable HDL code to an optimized cycle accurate C++/SystemC model[17]. This makes the simulation of large systems faster than commercial event-driven HDL simulators. This was a crucial choice for the experiments, since on other EDA tools such as Modelsim, the execution of a single Mibench program can take many days to complete.

Verilator produces a C++ class that models the system, either with C++ code or by wrapping a SystemC model, depending on the configuration. The class can be instantiated in a program that acts as testbench for the converted system. Inputs and outputs are exposed as 2-states variables (that can be either 0 or 1): after setting the input variables values, the testbench must call a special method, to evaluate the new outputs. Note that, since Verilator generates a cycle-accurate model, all intra-cycle delays are ignored.

While Verilator is supported natively by the FuseSoC project, it's a relatively new addition: the OR1200 Verilator testbench thus lacks many of the features provided by the default Verilog one bundled with the processor core. This testbench provides instruction tracing and disassembly, support for all the special `l.nop` codes and advanced logging, all functions that are indispensable for testing the modified OR1200: these feature have been added to a new Verilator testbench from scratch.

7.1.3 Terasic DE10-Lite

Albeit briefly, an FPGA was used to perform the final verification of the system. The Terasic DE10-Lite is an Altera (now Intel) MAX 10 based FPGA board. The board utilizes the maximum capacity MAX 10 FPGA, which has around 50K *logic elements* (LEs) and has a 64MB SDRAM.

Unfortunately, while FuseSoC provides ready to use configurations for many FPGA boards, the DE10-Lite isn't one of them. A similar board, the DE0-Nano (also by Terasic), is however supported and has been used as a base to develop a DE10-Lite port. Due to the emphasis on re-usability of the FuseSoc project, creating such port is only a matter of selecting the right configuration of external peripherals, to the match the ones provided by the new board, and parameters, such as clock frequency and external memory size.

7.2 Design verification

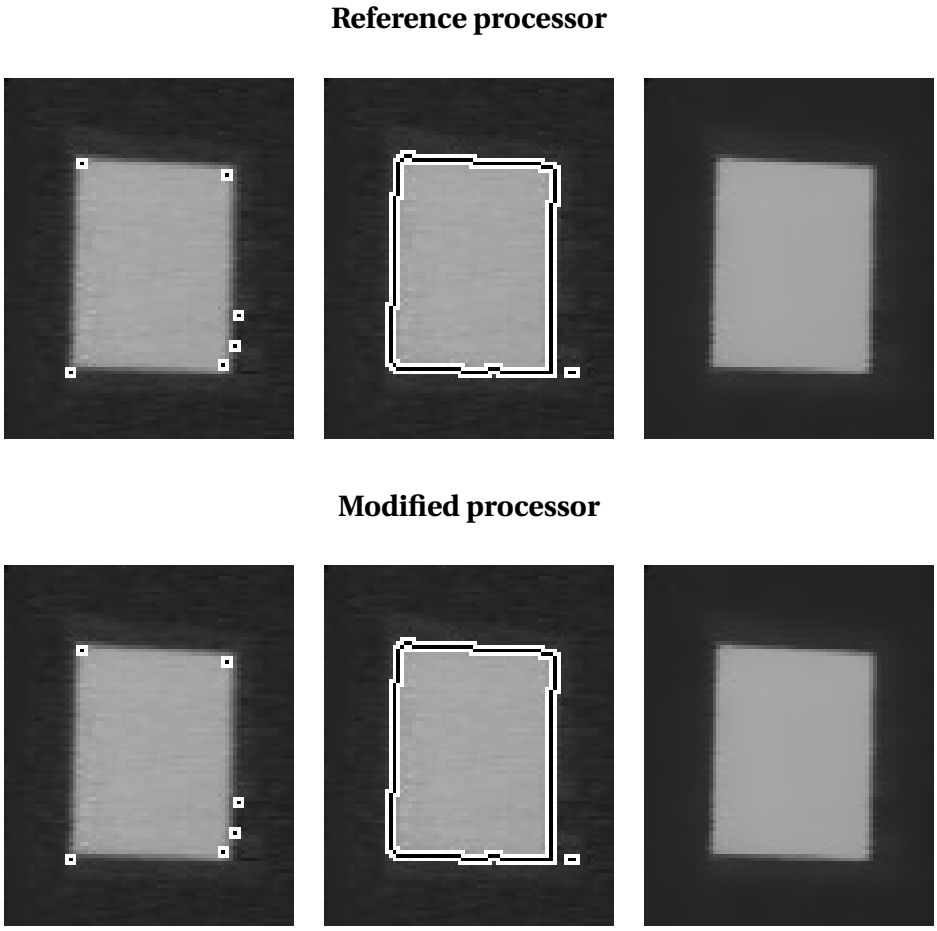

Figure 7.1: Mibench "susan small" output comparison

The processor was extensively tested to ensure the equivalence between the modified and unmodified version in the execution of sample programs. The design verification was performed in two steps, by respectively comparing:

1. Benchmark outputs

2. Instruction traces

The first approach consists in executing the same program on the reference processor, and then on the modified one, to check that the output of the benchmark is the same. This method is extremely simple and very effective as a coarse verification method, since it doesn't allow to check the executed instructions and returns no debug information in case of error. The main appeal of this approach is that can be entirely performed on the synthesized processor on an FPGA, outputting the benchmark output through an UART interface, and thus can be executed at very high speed.

The second approach requires tracing the executed instructions on the reference/-modified processor, and verify that the substitutions are performed correctly. This allows for a more precise verification and returns useful informations in case of error, but must be performed using a logic simulator, and thus can be very slow. Moreover, depending on the amount of traced information, simulation output can be very difficult to handle due to the sizable amount of data generated. A simple Python script was developed to check the trace automatically: the programs parses the two output traces verifying that the GPRs and SPRs contain the same data in both, after each substitution.

The high performance provided by running the benchmark on FPGA allows to verify the execution of all select Mibench programs with both the small and large dataset, while using logic simulation only allows to test the small version. The trace method, due to its better insight in error, was used as a debug tool and the output method was instead used as a final further verification. The final obfuscator design has been verified with both techniques, proving its correct behavior.

While countless errors have been detected and promptly fixed, during this testing phase, no errors caused by faulty substitutions were detected, proving that the verification techniques used during the substitution table design were effective.

7.3 Logic synthesis

Both the original and modified processor have been synthesized using Synopsis Design Compiler to estimate the area/delay impact of the obfuscator. The NanGate 45nm Open Cell Library[18], an open-source standard-cell library, has been used for this process. Synthesis results for the reference processor, and the modified one are presented respectively in table 7.1 and 7.2. It is possible to notice that the on-chip obfuscator has a longer critical path compared to the reference design: unsurprisingly the additional delay is introduced by the substitutions generation logic, i.e. the chain index generator, LUT, and instruction generator. When synthesized for 100 MHz the modified processor has an increased area equivalent of 2314 Nand2 logic gates (6.30% area increase). Of this area overhead approximately 500 gates are occupied by the substitution LUT

(1.36% of the total area): it's thus possible to estimate that each additional LUT added to the library will contribute to a similar area increase on the design.

Clock (MHz)	**Slack** (ns)	**Area** (μm^2)	**Area** (Cells)
50	13.04	29297	36713
100	3.04	29278	36689
300	0.00	29850	37406
inf.	-1.85	32590	40840

Table 7.1: Reference OR1200 synthesis results

Clock (MHz)	**Slack** (ns)	**Area** (μm^2)	**Area** (Cells)	**Overhead**
50	10.45	31112	38988	6.19%
100	0.64	31124	39003	6.30%
300	0.00	32442	40654	8.68%
inf.	-3.13	35582	44589	9.17%

Table 7.2: Modified OR1200 synthesis results

7.4 Performance evaluation

To evaluate the performance of the on-chip obfuscator, a suitable test must be designed, in order to, not only estimate the metrics of the proposed method, but also be fair to both the attacker and the system designer. A naive approach could be mimicking Trojan insertion: design a simple Trojan and check if a malware program is able to activate it with the obfuscator active. This technique, while close to a real life attack scenario, is highly dependent on knowledge on the on-chip obfuscator capabilities, since it requires designing a trigger beforehand, and could lead to biased results either in favor of the obfuscator designer or the attacker.

To avoid designing a trigger a different method is proposed: a benchmark program is traced in order to obtain all the instructions executed on an unmodified OR1200, the reference processor. Within these instructions all the sequences that could be used as trigger, called *candidates*, are selected for analysis. These sequences are then searched within the instruction trace of the obfuscated program: if a trigger sequence is no longer present the obfuscation is successful. This method allows to simultaneously check many triggers at the same time, without designing an actual Trojan and taking into account real rare sequences of instructions.

The test procedure is thus divided into four separate steps:

1. Executing the benchmark program on the reference system
2. Finding trigger candidates in the reference instruction trace
3. Execute the benchmark program on the test system
4. Evaluate performance

Trigger candidates are sequences of operation within the reference instruction trace that meet the characteristics of a potential trigger sequence, as described in section 4.4.1: they are expected to be short, to reduce the complexity of the trigger circuit, and rare to avoid accidental activation. Are thus considered HT sequence candidates, the instruction sequences between 3 and 10 instructions long, that have less than 10 instances within the original program trace. Only the lower bound is critical in the experimental result, since it has been demonstrated that, with a probabilistic approach, a longer substitution are easier to eliminate. Maximum instance count is irrelevant for similar reasons, but serves the purpose of selecting realistic candidates to better simulate a real-life attack scenario.

Obfuscation performance is experimentally evaluated in terms of in terms of:

- Trojan survival rate
- Instruction count ratio
- Run-time ratio

The *Survival rate* is the ratio between the number of candidates that "survived" the obfuscation process and the total number of candidates in the reference program. *Instruction count ratio* (ICR) measures the increase in the number of executed instructions produced by the obfuscator, and is measured as the ratio between the instruction count in an obfuscated program and the reference program. Finally, *Run-time ratio* (RTR) is identical to ICR, but instead of considering instruction count, evaluates clock cycle to estimate the execution time increase.

All these three parameters are studied in function of the *Substitution Frequency* (f), i.e., the number of substitutions performed per executed instructions.

In the following sections are reported the experimental result for the obfuscator performance evaluation. All the test have been executed on Verilator using the "small" version of the selected Mibench program, except basicmath in which the custom "tiny" configuration was used.

7.4.1 Reference values

Table 7.3 reports the reference results for the selected Mibench programs, in terms of number of candidates found, executed instructions and run-time. Note that candidate

Test program	Candidates	Instr. count	Run-time
basicmath	26,191	5,701,116	13,448,938
bitcount	15,651	5,852,886	8,295,385
qsort	15,471	23,134,860	62,864,341
susan-corners	27,070	3,510,480	7,730,449
susan-edges	28,426	6,206,010	13,233,602
susan-smooth	25,070	32,954,485	64,001,718

Table 7.3: Mibench reference values

Test program	Branch	ALU	System	Flag	Memory
basicmath	14.86%	59.04%	2.56%	11.91%	11.61%
bitcount	16.97%	57.56%	3.64%	15.02%	6.78%
qsort	16.78%	33.49%	1.96%	12.89%	34.86%
susan-corners	13.14%	50.25%	3.34%	10.07%	23.17%
susan-edges	12.67%	53.13%	1.95%	10.56%	21.66%
susan-smooth	9.98%	62.63%	0.85%	9.28%	17.24%

Table 7.4: Mibench instruction profile

and instruction count appear to be uncorrelated: very long programs, such as qsort appear to have fewer candidates that shorter program like bitcount. While the overall number of instruction sequences in a program trace clearly increases with the trace length, in order to be a candidate, a sequence, must also be rare. This means that the number of candidates is highly dependent on the program structure, not on the program trace length.

Candidate distribution per sequence length in each program is shown in figure 7.2: as expected the number of candidates found increases with the sequence length. A longer sequence is in fact more rare, and thus more likely to be a candidate.

Finally, table 7.4 reports the percentage of instruction executed per type of each benchmark program.

7.4.2 Survival rate

As shown in figure 7.3, survival rate declines as substitution frequency f increases, with a consistent behavior among the different test programs. It is possible to notice that the trend is not strictly decreasing: this is because we consider a trigger eliminated only when all its instances are removed from the program trace. However, a trigger sequence can be reintroduced accidentally by the obfuscation pattern, only to be removed at higher values of f.

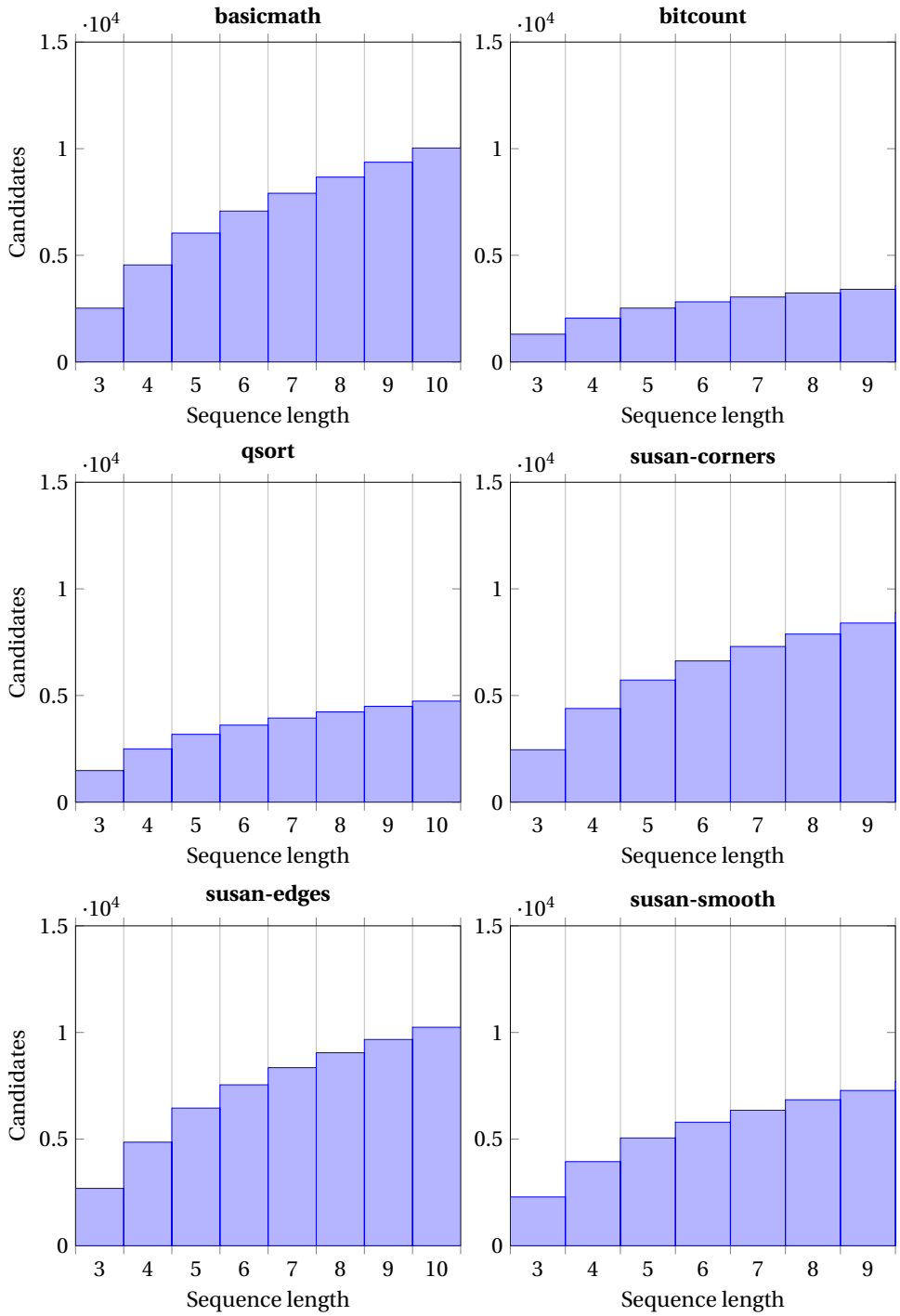

Figure 7.2: Candidates distribution with respect to sequence length

The average survival rate among all programs is presented in figure 7.4: notice that

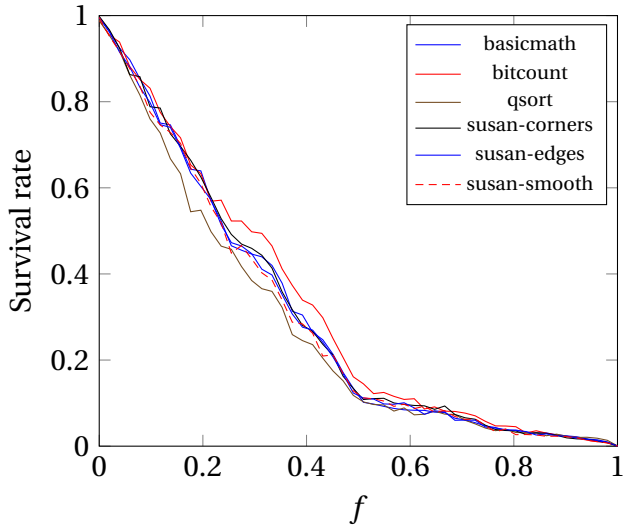

Figure 7.3: Survival rate in function of substitution frequency

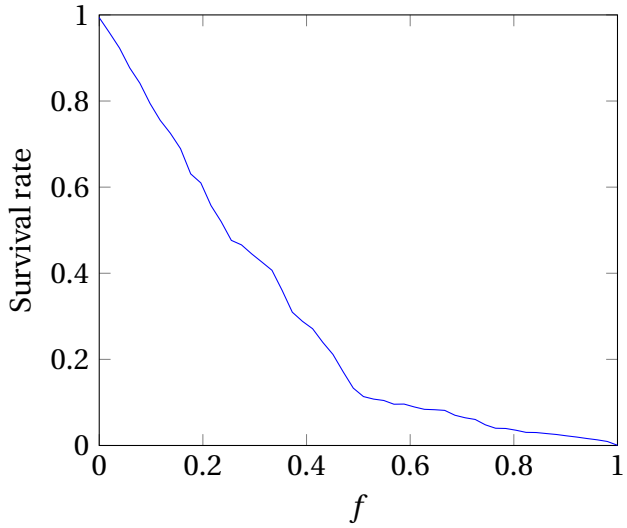

Figure 7.4: Average survival rate in function of substitution frequency

this pattern, while not strictly exponential, is somehow reminiscent of the one proposed in the efficiency model: the main difference between the twos is that, while the predicted trend was computed by evaluating triggers of a given substitution length independently, the experimental one takes into account multiple length concurrently.

Figure 7.5 reports the distribution of surviving candidates, classified by their length, at different values of substitution frequency in the program susan-small. As expected from the theoretical model, it can be clearly seen that longer candidates are eliminated at lower substitution frequencies, while shorter ones are more resilient.

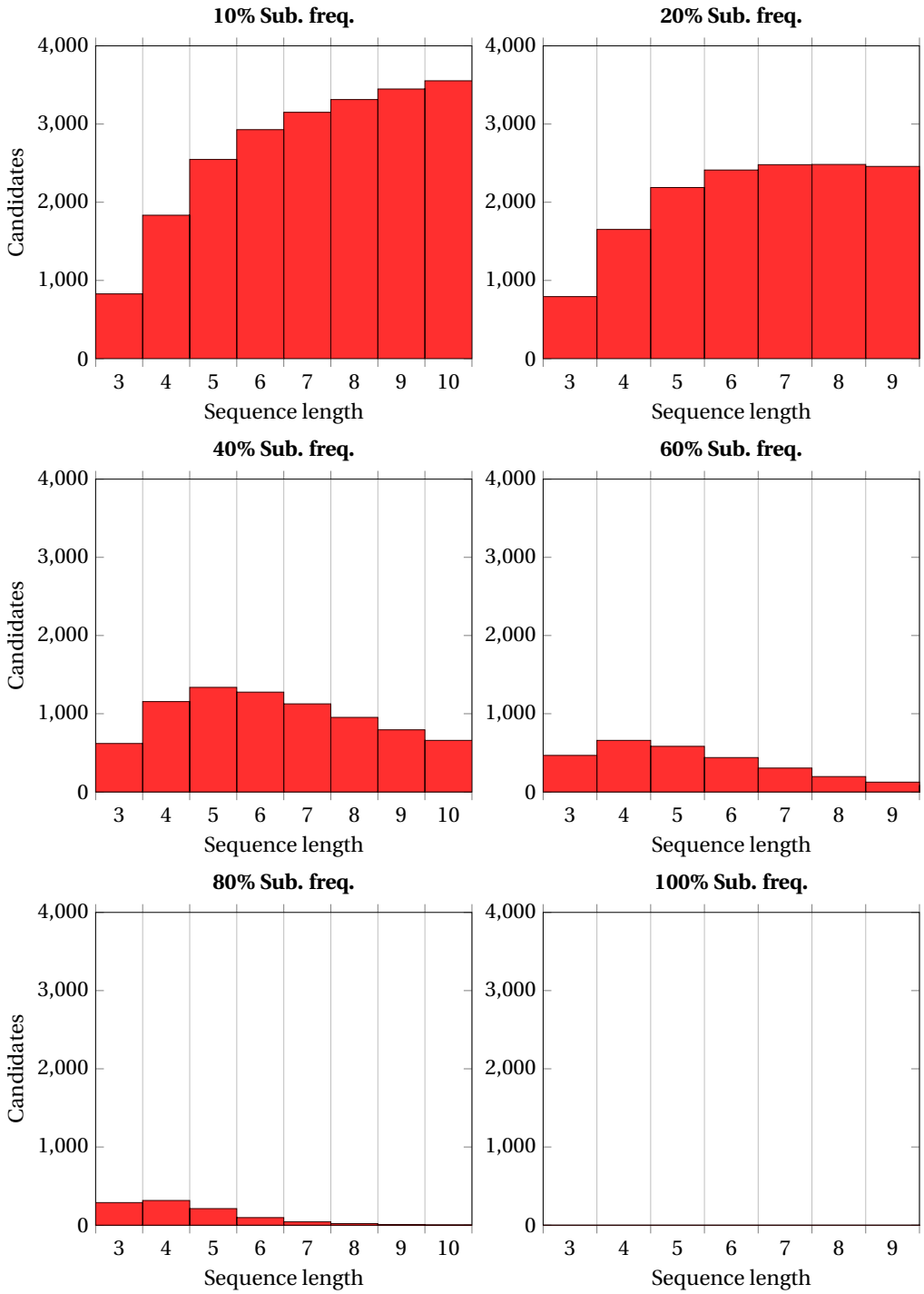

Figure 7.5: Survivor distribution at different substitution frequency values

Survivors

In all six benchmarks, substitution frequency of 100% instruction didn't achieve the complete elimination of all candidates: at the maximum value of f between 9 to 15 trigger sequences, with length between 3 and 4 instructions, were still present in the obfuscated code. In most cases, while these sequences appeared less than 10 times in the reference program trace, due to their selection as candidates, after obfuscation their number is increased up to hundreds, or in some cases even thousands of instances. This is phenomenon is caused by the obfuscation pattern itself: while some sequence may be very unusual in the reference program, they can be extremely frequent in the resulting substitution pattern. However, this instance explosion doesn't represent an issue, since if the instruction sequence becomes extremely frequent with the obfuscator active, it can't be used as trigger sequence.

Test program	**IBO**	**IAO**	**Sequence**
basicmath	10	10	`l.bf l.mul l.movhi`
	3	4	`l.sfgts l.bf l.xor`
	6	2	`l.movhi l.sfles l.bf`
bitcount	7	7	`l.bf l.mul l.mul`
	7	7	`l.bf l.mul l.mul l.mul`
qsort	7	7	`l.bf l.mul l.mul`
	7	7	`l.bf l.mul l.mul l.mul`
susan-corners	1	1	`l.jal l.mul l.sw`
susan-edges	1	1	`l.jal l.mul l.sw`
susan-smooth	2	2	`l.jal l.mul l.sw`

Table 7.5: List of critical survivors

To raise concerns are instead the sequences that are still present after obfuscation, whose number of instances didn't change or only changed slightly: these are the forsaken blind spots of the obfuscator. To highlight these particular class of sequences, here called *Critical survivors*, the survivors from each program were compared removing the one whose value exceeded the maximum instances to be considered a candidate in at least one of the programs after obfuscation. The results are reported in table 7.5, showing the number of *Instances before obfuscation* and *Instances after obfuscation*. It is clear, inspecting these sequences, that the instruction `l.mul` has no substitution in the current iteration of the substitution library: by adding a dummy substitution for this instruction all the related sequences are destroyed. Remaining sequences can be similarly removed by tuning the substitution table.

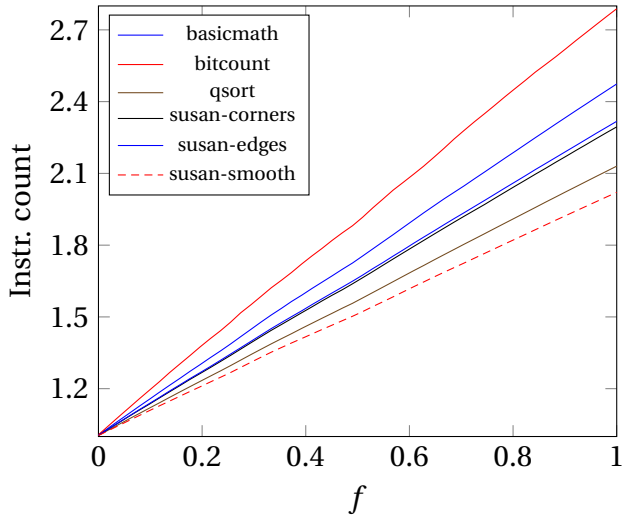

Figure 7.6: Instruction count in function of the substitution frequency

7.4.3 Instruction count ratio

Figure 7.6 shows that ICR grows linearly with respect to f, with a similar behavior in all Mibench programs. The angular coefficient of this linear trend varies from peaks of 2.78, for bitcount, and lows of 2.02, for susan-smooth. The average value for all trends is 2.33, a result surprisingly close to the average substitution length (2.31).

7.4.4 Run-time ratio

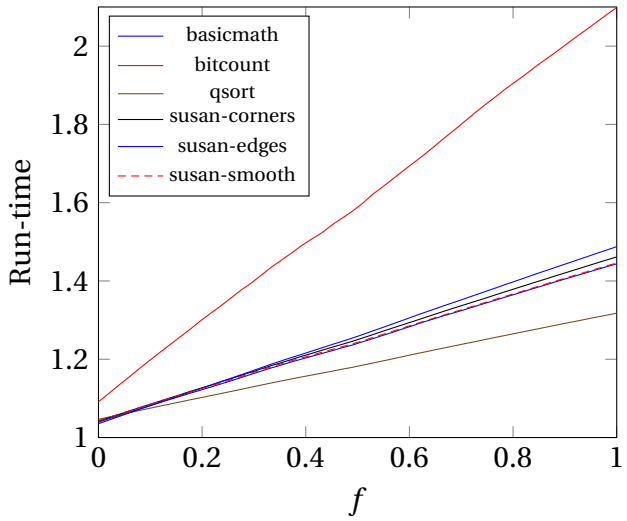

Figure 7.7: Run-time in function of the substitution frequency

Test program	Clock per Instr.
basicmath	2.35
bitcount	1.41
qsort	2.71
susan-corners	2.20
susan-edges	2.13
susan-smooth	1.94

Table 7.6: Mibench average clock cycles per instruction

Similarly to ICR, RTR increases linearly with f (figure 7.7). This time, while the trend is consistent and comparable for most benchmark program, bitcount has a very anomalous behavior.

Run-time is, in fact, expected to be highly dependent on the program instruction profile: if a program is composed by instructions with a high substitution length the run-time is heavily affected. Moreover, since instruction are dispatched during IF stalls, caching affects greatly run-time. While no information on instruction caching has been gathered in the proposed experiments, it can still be estimated with the ratio between clock cycles and instruction count. This parameter, reported in table 7.6, is only qualitative since it depends on multiple factors, such as data caching and the type of instructions executed.

Regardless of the qualitative nature of this estimation, run-time appears to be highly correlated to this value: programs with poor cache performance, such as *basicmath*, have a smaller run-time overhead than programs with better caching such as *bitcount*. Moreover, while the instruction profile of bitcount doesn't seem pretty anomalous, this particular program performs a large amount of bitwise operations, that, by change, have unusual long substitution length with respect to the average.

7.4.5 Optimal substitution frequency

Test program	Survival rate	ICR	RTR
basicmath	0.135	1.725	1.236
bitcount	0.161	1.881	1.578
qsort	0.118	1.557	1.179
susan-corners	0.134	1.638	1.245
susan-edges	0.124	1.649	1.254
susan-smooth	0,128	1,503	1.239

Table 7.7: Performance result for 50% obfuscation

Test program	Survival rate	ICR	RTR
basicmath	$0.573 \cdot 10^{-3}$	2.474	1.443
bitcount	$0.575 \cdot 10^{-3}$	2.788	2.098
qsort	$0.582 \cdot 10^{-3}$	2.130	1.317
susan-corners	$0.406 \cdot 10^{-3}$	2.294	1.461
susan-edges	$0.422 \cdot 10^{-3}$	2.318	1.487
susan-smooth	$0.480 \cdot 10^{-3}$	2.021	1.446

Table 7.8: Performance result for 100% obfuscation

Defining an optimal substitution frequency that is valid for all possible application of the modified processor is very difficult: substitution frequency affects both performance and run-time overhead, and thus its value is a trade-off between security and performance.

From the experimental data gathered, the substitution frequency of 50%, could represent a good trade-off: from what can be seen from picture 7.4, the average survival rate decreases steeply for the first half of the trend, then slowly converges to 0. This, combined with the consistent linear increase in run-time, means that increasing the substitution frequency after this value won't give the same improvement in efficiency than before, but would give the same increase in run-time.

Detailed results for substitution frequency of 50% and 100% are reported respectively in table 7.7 and 7.8: at a substitution frequency of 50%, the average survival rate is of 12.8% with a run-time increase of 23% while obfuscating all the instructions executed, the survival rate drops to almost zero and the run-time increases to 44%.

Chapter 8

Conclusion

8.1 Summary

This thesis investigated the use of an on-chip obfuscation technique to prevent the activation trough software of hardware Trojans injected in embedded processors. The proposed technique uses a simple, yet effective, rule based obfuscation pattern that substitutes single instructions with an equivalent sequence of operations before execution. Such pattern has been carefully studied, developing techniques to create and test substitutions as well as estimating their efficiency in obfuscating the code. A broad set of substitutions for the ORBIS32 instruction set was developed and verified using such techniques.

A prototype on-chip obfuscator was developed for the OR1200 open-source processor, to test the proposed solution. This modified processor employs a six-stage pipeline to obfuscate instructions after they are fetched from memory, before being decoded and consequently executed. The resulting design, while still lacking some features, has proven to be very reliable and requires minimum alteration to the original processor to be inserted.

Experimental results on a set of benchmark programs present a promising indication that the proposed obfuscation method could represent a viable way to mitigate the activation of injected HTs.

8.2 Limitations

The proposed obfuscation technique relies on a probabilistic approach to eliminate trigger sequences, and is thus far from being absolutely reliable: the attacker could fortuitously design a sequence that is unmodified by the obfuscator, or the trigger sequence could be generated by chance in the obfuscated code. However, while the attacker could be lucky in one instance, the configurable nature of the obfuscator means that the attacker cannot reliably target all manufactured processors.

In real-life scenarios, the attacker could exploit blind spots in the obfuscation pattern, or even use predictable obfuscation results, to activate a Trojan at will: the substitution library could however be tweaked to eliminate similar threats. Another limitation is imposed by the substitution library: designing a substitution for a reference instruction is a challenging endeavor, since absolute equivalence must be guaranteed. Moreover, due to their highly specialized nature, True Substitutions are hardly reusable in architectures other than the one for which they were created. The substitution design process, is time-consuming and it is tightly bounded to the target CPU architecture. New strategies should be developed to aid substitution design, and their execution on the processor.

Moreover, the overhead introduced by the obfuscator could be excessive in high-performance applications, especially at high substitution frequency, or if many LUTs are implemented. While the latter choice depends on the processor designer, the substitution frequency can be configured by the final user. Alterations to the current obfuscator architecture, such as a two stage obfuscation cycle, could potentially reduce the obfuscation latency, by pipelining the process: instruction indexing and pointer table read could be performed in the first clock cycle, while library read and instruction generation in the second.

It is worth to notice that the obfuscator is not immune to hardware piracy, and as such could still be compromised during manufacturing. However, if the attacker wants to activate a trigger via software in any stage other than the ID, he/she must always compromise the obfuscator. Post manufacturing testing should thus concentrate on verifying the trustworthiness of the ID and IOB stages alone. This is more convenient than verifying the trustiness of the entire processor, partially due to the simplicity of the two stages. Safety critical elements of the architecture, such as the obfuscator logic, could be designed with *Design for Hardware Trust* paradigm to aid their verification. Finally, in the proposed framework we considered a trigger eliminated if its code was no longer present in the obfuscated program trace. However, this analysis purposefully omitted to take into account that the obfuscated code could still produce internal signals similar, if not identical to the ones used to activate the Trojan. A possible future work should concentrate on assessing a substitution strength in these terms, as well as tweaking substitutions to be more effective in generating different signals values, data transaction and processors states.

8.3 More than Trojans

While writing this thesis two major papers were published that disclosed a whole class of architectural vulnerabilities affecting most modern processors: these papers are known as *Meltdown*[19] and *Spectre*[20]. In both cases out-of-order execution is used to leak the sensitive data and it is later retrieved using side-channel attacks. These vulnerabilities quickly reached mainstream audiences outside the academic world due to

the incredible number of processor potentially vulnerable: as of 2018, almost every computer system is affected, including desktops, laptops, and mobile devices. Specifically, Spectre has been shown to work on Intel, AMD, ARM-based, and IBM processors.

While the existence of similar vulnerabilities has never been questioned, the discovery of Meltdown and Spectre has stirred the research community and the public interest towards them. These exploits allow to carry out an attack very similar to the one described in the threat model used for this research, without compromising the underlying hardware, and as such this thesis wouldn't be complete without addressing these issues. Architectural attack patterns are attacks that exploit flaws in the architectural design of the system, such as weaknesses in protocols, authentication strategies, and system modularization. Exploiting intrinsic architectural vulnerabilities has many advantages with respect to HT injection. The attack doesn't require compromising the hardware, and as such can be carried at any point in the life time of the device, allowing for an extended time window for the attack to be devised and performed. While implementing similar attacks requires an extensive knowledge of the target architecture and hardware, in most cases minimal if no reverse engineering is required to implement them, especially if compared with Trojans, where the entire IC netlist has to be analyzed by the attacker. Despite Meltdown and Spectre target high-end processors, similar architectural exploits can used to attack embedded systems. This final section of the thesis will quickly dwell into these type of attacks.

8.3.1 Fault injection

Fault injection attacks are hardware attacks that induce errors in the target to leak confidential information either directly or trough side-channel. These type of attacks can be divided into *passive*, where the attack is performed without tampering by exploiting natural occurring errors, *semi-active*, where temperature, voltage or clock are manipulated to produce errors, and *active*, where the victim package is removed and light and lasers are used to produce errors. While many types of attacks can be implemented using fault injection, an interesting example is the RSA attack with CRT (Chinese Remainder Theorem), also known as the *Bellcore attack*[21]. In this type of attacks the attacker injects faults in the signature computation obtaining one or many faulty signatures, from which the private key can be retrieved.

Fault attacks typically require physical access to the device under attack to be performed and thus, at first glance, seem entirely relegated to the hardware world. However, they have been carried out via software using the Rowhammer attack[22]. Rowhammer is an unintended side effect in *dynamic random-access memory* (DRAM), where repeated accesses to a given memory line produces electrical interaction with the neighboring ones. This type of attack allows to indirectly read and manipulate memory otherwise inaccessible, with potentially catastrophic effects. By repeatedly accessing one or more memory rows (aggressor rows) via software, the attacker can produce a bit flip

in a neighboring row at will. This attack mechanism is counteracted by the memory refresh cycle, and as such, performing a Rowhammer attack requires fast access to the DRAM, bypassing the memory hierarchy, and knowledge on the physical topology of the memory used.

8.3.2 Side-channel attacks

Side-channel attacks are attacks that use information gained by observing the implementation, rather then rather than weaknesses in the implemented algorithm itself. The most commons side-channels are timing and power. As in the case of fault injection these type of attacks are most commonly implemented at the physical level, but they can be also implemented and performed successfully via software.

Cache attack

A computer architecture design aims at optimizing processing speed and, as such, the execution time and memory access patterns of a program are strongly correlated to the data being processed. When performing attacks against well known algorithms, such as cryptographic algorithms, measuring these parameter can be used to indirectly determine the data being processes.

Since cache is strongly related to both execution and processing time, and lacks any access restrictions, it is the perfect candidate for this type of attack. In fact, a malware program can monitor it is own cache access times to determine which data was modified by the victim program (access attack), or by studying the victim hit/miss pattern (timing attack). These attacks have been proven successful against RSA, AES and OpenSSL, allowing the attacker to obtain sensitive data knowing only the algorithm implementation and the state of the cache before its execution. Bernstein in [23], demonstrates a complete AES key recovery using this technique.

Branch prediction attack

Similarly to cache, a Branch Prediction Unit content is strongly related to execution, thus a skilled attacker can obtain precious informations about the data being processed by a victim program, if its execution flow of highly dependent on the secret data. In [24] an branch prediction attack capable of breaking RSA encryption was presented: while the proposed approach required a CPU capable of simultaneous multi-threading for the attack to be performed, the authors are positive that it is only a matter of time before similar vulnerabilities can be exploited on other architectures.

8.4 Final notes

After the design and testing of the obfuscator, its limitations remain mostly unchanged with respect to the ones highlighted in the preliminary analysis: on-board obfuscation is a viable technique to prevent hardware Trojan activation, at least for the proposed threat model, however the drawback introduced by the additional hardware could be unsuitable for most applications. This combined with other concerns such as the scope of the chosen threat model and the existence of other effective attacks other then HT injection make the perspective of using such device in a commercial processor very unrealistic: even if the application of a processor could allow the performance drop caused by the obfuscator, such device can only protect from a small class of Trojans, and thus may not be considered worth the effort. While activating a trigger using instruction sequences is an interesting approach, there are many other ways to activate a Trojan.

Side-channel based Trojan detection techniques have their fair share of disadvantages, but they can potentially highlight any type of alterations to an IC: any Trojan design, no matter how unusual or ingenious can be detected by these techniques. On the other hand, architecture based solution can only defend the processor from known threats, drastically limiting their scope and effectiveness. In conclusion, side-channel analysis is, and will be for a long time, the best resource against hardware piracy.

Lately the existence itself of Hardware Trojans have been put into question: while multiple news sources claim that some unexpected hardware failures or data-leakages are caused by hardware-piracy attacks, no definitive evidence exists to back these claims[25]. Injecting a hardware Trojan is difficult, time-consuming and expensive: while hardware Trojans represent a real threat to the current IC market, they have never been spotted in the wild.

Appendix A

ORBIS32 Instruction set

Instruction	Type	Index	Description
l.add	A	64	Addition
l.addc	A	65	Addition with carry
l.addi	I	27	Immediate Addition
l.addic	I	28	Immediate Addition with carry
l.and	A	67	Bitwise and
l.andi	I	29	Immediate bitwise and
l.bf	B	2	Branch if flag is set
l.bnf	B	3	Branch if flag is not set
l.div	A	79	Division signed
l.divu	A	80	Division unsigned
l.extbs	A	60	Signed byte extend
l.extbz	A	63	Zero byte extend
l.exths	A	58	Signed half-word extend
l.exthz	A	62	Zero half-word extend
l.extws	A	59	Signed half-word extend
l.extwz	A	61	Zero half-word extend
l.ff1	A	71	Find first 1
l.fl1	A	76	Find last 1
l.j	B	0	Jump
l.jal	B	1	Jump and link
l.jalr	M	13	Register jump and link
l.jr	M	14	Register jump
l.lbs	I	24	Load byte signed
l.lbz	I	23	Load byte unsigned
l.lhs	I	26	Load half-word signed
l.lhz	I	25	Load half-word unsigned
l.lws	I	22	Load word signed

Instruction	Type	Index	Description
l.lwz	I	21	Load word unsigned
l.mac	N	49	Multiply and accumulate
l.maci	N	15	Immediate multiply and accumulate
l.macrc	N	6	MAC read and clear
l.mfspr	I	33	Move from SPR
l.movhi	I	5	Move to m.s. half-word
l.msb	N	51	Multiply and subtract
l.mtspr	M	48	Move to SPR
l.mul	A	77	Multiply signed
l.muli	I	32	Immediate multiply signed
l.mulu	A	81	Multiply unsigned
l.nop	S	4	No operation
l.or	A	68	Bitwise or
l.ori	I	30	Immediate bitwise or
l.rfe	I	12	Return from exception
l.ror	A	75	Rotate
l.rori	I	37	Immediate rotate
l.sb	M	56	Store byte
l.sfeq	F	83	Set flag if equal
l.sfeqi	FI	38	Set flag if equal immediate
l.sfges	F	90	Set flag if greater/equal signed
l.sfgesi	FI	45	Set flag if greater/equal signed immediate
l.sfgeu	F	86	Set flag if greater/equal unsigned
l.sfgeui	FI	41	Set flag if greater/equal unsigned immediate
l.sfgts	F	89	Set flag if greater signed
l.sfgtsi	FI	44	Set flag if greater signed immediate
l.sfgtu	F	85	Set flag if greater unsigned
l.sfgtui	FI	40	Set flag if greater unsigned immediate
l.sfles	F	92	Set flag if lower/equal signed
l.sflesi	FI	47	Set flag if lower/equal signed immediate
l.sfleu	F	88	Set flag if lower/equal unsigned
l.sfleui	FI	43	Set flag if lower/equal unsigned immediate
l.sflts	F	91	Set flag if lower signed
l.sfltsi	FI	46	Set flag if lower signed immediate
l.sfltu	F	87	Set flag if lower unsigned
l.sfltui	FI	42	Set flag if lower unsigned immediate
l.sfne	F	84	Set flag if not equal
l.sfnei	FI	39	Set flag if not equal immediate
l.sh	M	57	Store half-word
l.sll	A	72	Shift left logically

Instruction	Type	Index	Description
l.slli	I	34	Immediate shift left logically
l.sra	A	74	Shift right arithmetically
l.srai	I	36	Immediate shift right arithmetically
l.srl	A	73	Shift right logically
l.srli	I	35	Immediate shift right logically
l.sub	A	66	Subtract
l.sw	M	55	Store word
l.sys	S	7	System call
l.trap	S	8	Trap
l.xor	A	69	Bitwise xor
l.xori	I	31	Immediate bitwise xor

Appendix B

Substitution library

Reference	Jacc.	SMD	Substitution
l.add rD,rA,rB	0.50	0.04	l.addi r0,r0,0 l.addc rD,rA,rB
l.addc rD,rA,rB	0.17	0.002	l.addc rD,rA,rB l.and r0,r0,r0
l.addi rD,rA,I	0.42	0.04	l.add r0,r0,r0 l.addic rD,rA,I
l.addic rD,rA,I	0.33	0.05	l.xor rD,rA,r0 l.xori r0,r0,I l.addc rD,rD,r0 l.andi r0,r0,0
l.and rD,rA,rB	0.40	0.07	l.and rD,rA,rB l.extbs r0,r0
l.andi rD,rA,K	0.53	0.10	l.xori rD,rA,-1 l.ori r0,r0,K l.xori r0,r0,-1 l.or rD,rD,r0 l.xori rD,rD,-1 l.xor r0,r0,r0
l.extbs rD,rA	0.50	0.07	l.slli rD,rA,24 l.srai rD,rD,24
l.extbz rD,rA	0.50	0.07	l.slli rD,rA,24 l.srli rD,rD,24
l.exths rD,rA	0.50	0.07	l.slli rD,rA,16 l.srai rD,rD,16
l.exthz rD,rA	0.83	0.19	l.andi rD,rA,65535
l.extws rD,rA	0.40	0.07	l.xori rD,rA,0

Reference	Jacc.	SMD	Substitution
l.extwz rD,rA	0.40	0.07	l.xori rD,rA,0
l.ff1 rD,rA	0.36	0.09	l.ff1 rD,rA l.movhi r0,0
l.fl1 rD,rA	0.36	0.09	l.fl1 rD,rA l.movhi r0,0
l.lbs rD,I(rA)	0.51	0.17	l.lbz rD,I(rA) l.extbs rD,rD
l.lbz rD,I(rA)	0.48	0.13	l.lbs rD,I(rA) l.extbz rD,rD
l.lhs rD,I(rA)	0.51	0.17	l.lhz rD,I(rA) l.exths rD,rD
l.lhz rD,I(rA)	0.36	0.09	l.lhz rD,I(rA) l.exths rD,rD
l.lws rD,I(rA)	0.20	0.07	l.lwz rD,I(rA)
l.lwz rD,I(rA)	0.20	0.07	l.lws rD,I(rA)
l.mfspr rD,rA,K	0.43	0.11	l.mfspr rD,rA,K l.andi r0,r0,0
l.movhi rD,K	0.30	0.06	l.movhi rD,K l.xor r0,r0,r0
l.mtspr rA,rB,K	0.50	0.11	l.mtspr rA,rB,K l.andi r0,r0,0
l.nop K	0.67	0.12	l.movhi r0,0 l.nop K l.andi r0,r0,0
l.or rD,rA,rB	0.40	0.07	l.or rD,rA,rB l.andi r0,r0,0
l.ori rD,rA,K	0.40	0.06	l.xori rD,rA,-1 l.xori r0,r0,K l.exthz r0,r0 l.xori r0,r0,-1 l.and rD,rD,r0 l.xori rD,rD,-1 l.xor r0, r0, r0
l.ror rD,rA,rB	0.40	0.07	l.ror rD,rA,rB l.andi r0,r0,0

Reference	Jacc.	SMD	Substitution
l.rori rD,rA,L	0.53	0.09	l.xor rD,rA,r0
			l.xori r0,r0,L
			l.xori r0,r0,31
			l.addi r0,r0,33
			l.sll r0,rD,r0
			l.srli rD,rD,L
			l.or rD,rD,r0
			l.xor r0,r0,r0
l.sb I(rA),rB	0.50	0.17	l.sb I(rA),rB
			l.movhi r0,0
l.sfeq rA,rB	0.50	0.11	l.sfeq rA,rB
			l.ff1 r0,r0
l.sfeqi rA,I	0.77	0.12	l.xori r0,r0,I
			l.sfeq rA,r0
			l.xor r0,r0,r0
l.sfges rA,rB	0.40	0.07	l.sfles rB,rA
l.sfgesi rA,I	0.69	0.20	l.xori r0,r0,I
			l.sfges rA,r0
			l.xor r0,r0,r0
l.sfgeu rA,rB	0.50	0.07	l.sfleu rB,rA
l.sfgeui rA,I	0.69	0.16	l.ori r0,r0,I
			l.sfgeu rA,r0
			l.xor r0,r0,r0
l.sfgts rA,rB	0.50	0.07	l.sflts rB,rA
l.sfgtsi rA,I	0.70	0.17	l.xori r0,r0,I
			l.sfgts rA,r0
			l.xor r0,r0,r0
l.sfgtu rA,rB	0.67	0.07	l.sfltu rB,rA
l.sfgtui rA,I	0.71	0.14	l.ori r0,r0,I
			l.sfgtu rA,r0
			l.xor r0,r0,r0
l.sfles rA,rB	0.40	0.07	l.sfges rB,rA
l.sflesi rA,I	0.69	0.20	l.xori r0,r0,I
			l.sfles rA,r0
			l.xor r0,r0,r0
l.sfleu rA,rB	0.50	0.07	l.sfgeu rB,rA
l.sfleui rA,I	0.69	0.16	l.ori r0,r0,I
			l.sfleu rA,r0
			l.xor r0,r0,r0
l.sflts rA,rB	0.50	0.07	l.sfgts rB,rA

Reference	Jacc.	SMD	Substitution
l.sfltsi rA,I	0.70	0.17	l.xori r0,r0,I l.sflts rA,r0 l.xor r0,r0,r0
l.sfltu rA,rB	0.67	0.17	l.sfgtu rB,rA
l.sfltui rA,I	0.71	0.14	l.ori r0,r0,I l.sfltu rA,r0 l.xor r0,r0,r0
l.sfne rA,rB	0.50	0.13	l.sfne rA,rB l.ff1 r0,r0
l.sfnei rA,I	0.72	0.15	l.xori r0,r0,I l.sfne rA,r0 l.xor r0,r0,r0
l.sh I(rA),rB	0.50	0.17	l.sh I(rA),rB l.andi r0,r0,0
l.sll rD,rA,rB	0.40	0.07	l.sll rD,rA,rB l.andi r0,r0,0
l.slli rD,rA,L	0.63	0.11	l.xor rD,rA,r0 l.xori r0,r0,I l.sll rD,rD,r0 l.xor r0,r0,r0
l.sra rD,rA,rB	0.40	0.07	l.sra rD,rA,rB l.andi r0,r0,0
l.srai rD,rA,L	0.63	0.11	l.xor rD,rA,r0 l.xori r0,r0,I l.sra rD,rD,r0 l.xor r0,r0,r0
l.srl rD,rA,rB	0.40	0.07	l.srl rD,rA,rB l.andi r0,r0,0
l.srli rD,rA,L	0.63	0.11	l.xor rD,rA,r0 l.xori r0,r0,I l.srl rD,rD,r0 l.xor r0,r0,r0
l.sub rD,rA,rB	0.40	0.07	l.sub rD,rA,rB l.extws r0,r0
l.sb I(rA),rB	0.50	0.17	l.sb I(rA),rB l.movhi r0,0
l.sh I(rA),rB	0.50	0.20	l.sh I(rA),rB l.movhi r0,0
l.sw I(rA),rB	0.50	0.07	l.sw I(rA),rB l.movhi r0,0

Reference	Jacc.	SMD	Substitution
l.xor rD,rA,rB	0.40	0.07	l.xor rD,rA,rB l.rori r0,r0,0
l.xori rD,rA,I	0.35	0.06	l.xor rD,rA,r0 l.ori r0,r0,I l.exths r0,r0 l.xor rD,rD,r0 l.andi r0,r0,0

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 20, no. 3, pp. 33–35, 2006.

[2] E. Times, "Report: Tsmc's 3nm fab could cost $20 billion," https://www.eetimes.com/document.asp?doc_id=1332419, October 2017.

[3] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[4] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 15–19, 2008.

[5] G. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans: Extended version," vol. 4, pp. 19–31, 04 2014.

[6] J.-F. Gallais, J. Großschädl, N. Hanley, M. Kasper, M. Medwed, F. Regazzoni, J.-M. Schmidt, S. Tillich, and M. Wójcik, "Hardware trojans for inducing or amplifying side-channel leakage of cryptographic software," *Trusted Systems Lecture Notes in Computer Science*, pp. 253–270, 2011.

[7] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test*, pp. 1–1, 2013.

[8] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*. IEEE, 2008, pp. 51–57.

[9] H. Salmani, M. Tehranipoor, and J. Plusquellic, "New design strategy for improving hardware trojan detection and reducing trojan activation time," *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009.

[10] M. Banga and M. S. Hsiao, "Vitamin: Voltage inversion technique to ascertain malicious insertions in ics," *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009.

[11] "OR1200 processor implementation," https://github.com/openrisc/or1200.

[12] "Fusesoc," https://github.com/olofk/fusesoc.

[13] "Newlib," https://www.sourceware.org/newlib/.

[14] S. Bhunia and M. M. Tehranipoor, *The Hardware Trojan War: Attacks, Myths, and Defenses*. Springer, 2017.

[15] "Efficient shift registers, lfsr counters, and long pseudo-random sequence generators," https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf.
[16] "Mibench," http://vhosts.eecs.umich.edu/mibench/.
[17] "Verilator," https://www.veripool.org/wiki/verilator.
[18] J. Knudsen, "Nangate 45nm open cell library."
[19] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.
[20] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.
[21] A. Sidorenko, J. Van Den Berg, and R. Foekema, "Bellcore attack in practice," 03 2018.
[22] A. Fournaris, L. Pocero Fraile, and O. Koufopavlou, "Exploiting hardware vulnerabilities to attack embedded system devices: a survey of potent microarchitectural attacks," *Electronics*, vol. 6, no. 4, p. 52, Jul 2017. [Online]. Available: http://dx.doi.org/10.3390/electronics6030052
[23] D. J. Bernstein, "Cache-timing attacks on aes," 2005.
[24] O. Acıiçmez, C. Koc, and J.-P. Seifert, "Predicting secret keys via branch prediction," pp. 225–242, 02 2007.
[25] S. Adee, "The hunt for the kill switch," https://spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch, May 2008.