Turning electronic circuits features into on-chip locks

Brice Colombier, Lilian Bossuet and David Hély

Abstract In order to fight against counterfeiting and illegal copying of integrated circuits (ICs) and intellectual property (IP) cores, several design data protection schemes have been proposed. One of the key-component of such schemes is the one in charge of locking the circuit in case it has been illegally obtained. This is necessary in order to make illegal copies useless. In this chapter, we show that common features, found in most electronic devices, can be turned into on-chip locks. First of all, we identify these features, and then show how they can be modify to lock the design; The main point to use existing features is to induce low overhead, which is very interesting for designers. We implemented the proof of concept of the described locks in FPGAs, and present resources overhead for the implementation of them on two reference designs. We also give details on partial locking, which can be used to provide the design in evaluation mode.

1 Introduction and context

Following Moore's law, electronic systems are increasingly complex and powerful. Their complexity is following a similar trend, forcing designers to adopt a modular approach when designing such systems. Thus a design-and-reuse approach is followed, in which functional building blocks are put together by system integrators. These blocks are provided by IP cores designers, who must transfer their complete

Brice Colombier
Hubert Curien Laboratory, UMR CNRS 5516, University of Lyon, Saint-Étienne - France e-mail: b.colombier@univ-st-etienne.fr

Lilian Bossuet
Hubert Curien Laboratory, UMR CNRS 5516, University of Lyon, Saint-Étienne - France e-mail: lilian.bossuet@univ-st-etienne.fr

David Hély
LCIS, Grenoble Institute of Technology, Valence, France e-mail: david.hely@lcis.grenoble-inp.fr

design in order to have it implemented correctly. However, such a situation necessarily leads to abuses, since the designer can not control the number of instances implemented from its original design. It results in overbuilding IP cores and counterfeiting of integrated circuits, and the trend is growing. Multiple cases have been reported in recent years [1, 2, 3].

In order to answer this issue, the circuit can be provided as initially locked. It is then non-functional, and should be unlocked in order to be used. The unlocking procedure can be initiated only by the designer, allowing precise audit of the number of instances of the protected design. This is referred to as hardware metering [4]. In case the design has been obtained illegally, either from overbuilding or counterfeiting, it remains locked and therefore unusable.

There are several ways to achieve locking of a circuit. Among them, modifying the combinational logic is a way. It is presented in details in Chapter 3.

Another interesting approach is to look into hardware Trojans [5] that target Denial-Of-Service attacks. Such attacks are closely related to the kind of remote locking we want to achieve here. Therefore, the actual means that are used by hardware Trojans to trigger and achieve a Denial-Of-Service attack could be turned into remote locking techniques. It effectively turns malicious hardware into salutary hardware [6]. In order to be worth considering from the designer's perspective, a design protection scheme must also be cheap in terms of additional hardware resources required to implement it. Indeed, if the economic losses associated with the illegal actions are actually less expensive than the protection scheme itself, the latter becomes unsuitable. Therefore, the protection scheme should be as lightweight as possible, and occupy a very small area on the protected chip. This characteristic is very common for Hardware Trojans, which are usually found in the form of tiny and stealthy modifications of the original design.

The point here is to achieve locking by targeting very sensitive components. These components should be crucial to the proper functioning of the system. Thus their disablement will render the system absolutely unusable, indeed achieving locking of the whole design. These points of action can be thought of as *single points of failure*, which correct behaviour is absolutely necessary to the overall system.

In this chapter, we show that such points can be found in the vast majority of complex electronic systems [7]. This is valuable since the design protection scheme should be usable for any type of design, and should not depend on specific design features.

This chapter is organised as follows. Section 2 identifies the features which could be turned into on-chip locks, and provides a comparison of them using several criteria. Section 3 shows how such features can be modified to disturb the circuit's operation. Section 4 gives implementation results on FPGA. Two reference designs and two FPGA families were used. Section 5 proposes a discussion on partial locking, which is an interesting way to provide a circuit in evaluation mode.

2 Features usable as locking means

Figure 1 shows a complex electronic system. We highlighted the following common features which can be turned into on-chip locks:

1. The clock circuitry,
2. The inputs/outputs,
3. The processor,
4. The interconnection buses,
5. The system controller,
6. The analog components.

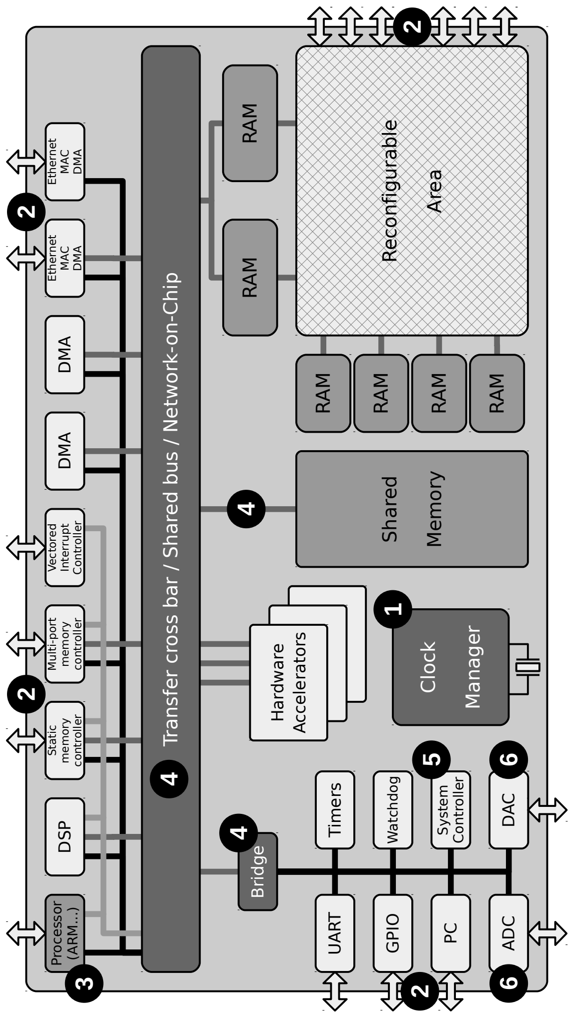

Fig. 1 SoC features which can be turned into on-chip locks

2.1 Clock circuitry

The first feature that is immediately identifiable as a locking point is the clock circuit. Indeed, it is a universal feature found in most digital designs. Moreover, good operation of the circuit is heavily dependent on the clock signal. Thus by acting on the clock signal, it is possible to disable the circuit, making it effectively locked. Another interesting characteristic of the clock is that its frequency is related to the device's performances. Hence by dynamically shifting the clock frequency it is possible to alter the performances of the circuit. This could be used to provide an evaluation version of the device, operating at a lower frequency and exhibiting a lower level of performance.

2.2 Inputs/outputs

All electronic designs have input and output ports to interact with other components. By temporarily disabling these ports, it is possible to prevent new data to be sent to the design. Even though it does not make the design unusable itself, it makes it almost useless, since it is then not possible to interact with it anymore.

2.3 Processor

When a processor is present in a digital design, it is usually a central component. Such processor can be either hardwired or soft-core. A soft-core processor is described in a hardware description language and implemented in reconfigurable resources. In essence, the processor executes a sequence of instruction. One way to alter its functionality is then to prevent the execution of new instructions.

2.4 Buses

Interconnection buses are the backbone of complex systems. They allow multiple IP cores to communicate. The integrity of the information exchanged between the different sub-modules of a system is a crucial requirement. Therefore, by altering this information, it is possible to render the system non-functional.

2.5 System controller

The control logic of complex designs is usually handled by an FSM. By modifying this FSM's states, it is possible to alter the operation of the circuit. Another possibility is to add extra states to control access to the normal mode of operation.

2.6 Analog components

In order to handle physical data, a design can integrate analog components. Such components are precisely calibrated to suit the needs of the designer. By altering this calibration, their behaviour can be altered.

Another important analog component of the design is the power supply module. By shutting down specific areas of the design, they can be efficiently disabled. This feature is called *power gating*, and is already implemented in some designs to reduce power consumption.

2.7 Global comparison

After identifying these features, we can have a first overview of their pros and cons. Table 1 presents a qualitative comparison.

The first criterion used to evaluate the features is the impact on performance. It describes how the performance of the circuit is affected during normal operation. Modifying the clock circuitry has low impact on the performance, although the clock characteristics such as the jitter can be affected if the modification is poorly handled. Acting on the inputs/outputs, the FSM or the processor does not have any impact on the circuit's performance. Conversely, modifying the buses can lead to slower data rates and increase the latency. Similarly, modifying the calibration of analog components can reduce their efficiency.

The second criterion is the ease of locking/unlocking. It quantifies how simple it is to implement locking using the corresponding feature. It also shows how easy it is to fall back into normal behaviour after an unlocking request has been received. For example, acting on the clock circuitry or the inputs is simple. They can be easily disabled and enabled again. On the other hand, modifying the processor to be able to stop it can be complicated. Similarly, tampering with the buses can lead to unexpected behaviour. In both cases, correctly coming back to normal behaviour might not be guaranteed. When modifying the FSM, locking refers to entering "hidden" states, corresponding to altered operation. Therefore, locking or unlocking requires to have access to the FSM inputs. This is not guaranteed, and most designs do not allow to transition between FSM states so easily. Similarly, modifying the calibration of analog components can be hard to achieve.

When modifying a feature to achieve locking, the impact on the circuit functionality should be as high as possible, to make is completely unusable. Disabling the clock, the processor, the buses or entering “hidden” FSM states systematically leads to complete locking. On the other hand disabling the inputs/outputs or altering the characteristics of analog components has medium impact, which will depend on the usage.

Finally, partial locking is possible with some of the described features. We define partial locking as a state in which the design has a correct behaviour, but has a lower level of performance. This is achievable only by acting on the clock or the analog components. Modifying the clock frequency directly affects the designs performance. Likewise, altering the calibration of analog components can make them perform poorly.

The final column on the right of Table 1 gives an overall suitability estimation for the feature. It estimates how suited the feature is in order to be turned into an on-chip lock.

Table 1 Qualitative comparison of the presented features when being used as on-chip locks

	Evaluation criterion				
Feature modified for locking	Impact of the locking scheme on performance	Ease of dynamic (un)locking	Efficiency/Impact on functionnality	Partial locking	Overall suitability
Clock	Low	High	High	yes	●●●
Inputs/outputs	None	High	Medium	no	●●○
Processor	None	Medium	High	no	●●○
Buses	Medium	Medium	High	no	●○○
FSM	None	Low	High	no	●○○
Analog parts	Medium	Low	Medium	yes	●○○

3 Practical transformation into on-chip locks

We now give means how to turn the features presented in Section 2 into on chip-locks. Analog components modification is not discussed here.

3.1 Clock circuitry

In practise, acting on the clock circuitry can be achieved in two ways. The first one is to use a modified clock-gating module. The second one makes use of the reconfiguration capabilities of some phase-locked loops (PLLs).

3.1.1 Clock gating

An already approved method to act on the clock is clock-gating. It is commonly used to reduce power consumption by not clocking the unused regions of the circuit. Therefore it could also be used to make the circuit unusable.

To achieve clock-gating, we insert a specific module on the clock signal path. This module is shown in Figure 2. It does not require to add extra logic on the clock signal path itself, but rather makes use of the clock-enable inputs of existing clock buffers.

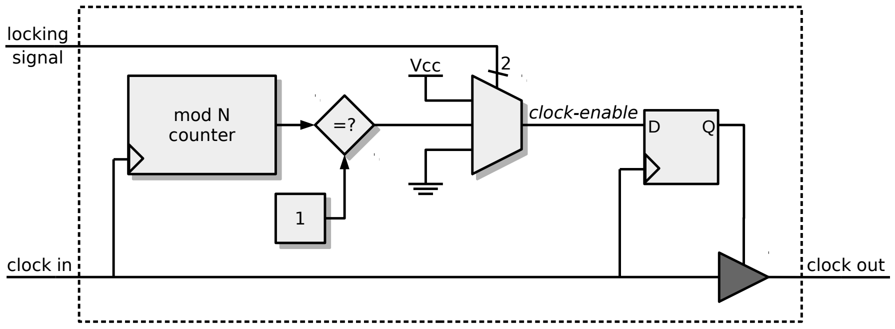

Fig. 2 Clock-gating module acting on a clock buffer (in dark-grey)

Here, the clock-enable input can be driven by three different signals. The first one, which corresponds to a high logic level (V_{cc}), allows to leave the clock signal unchanged. In this case, the circuit is totally unlocked. The second one is the result of the comparison between the output of an n-bit counter and the n-bit value 1. Thus the clock-buffer is only active when the counter is equal to 1. In practical terms, the output frequency is then divided by 2^n, where n is the size of the counter.

In this case, the output clock does not have a 50% duty cycle. Instead, the duty cycle α obtained from the division is given in Equation 1.

$$\alpha = \frac{t_H}{T} = \frac{t/2}{n.t} = \frac{1}{2n} \tag{1}$$

In such case, if the frequency is chosen to be divided by a large number, the duty cycle can drop to low values. However, if the setup times were not violated with the original frequency, then they will not be either with the divided frequency. The waveforms obtained from the three different modes presented here are shown in Figure 3. Figure 3a shows the original clock. Figure 3b shows the divided clock. Here, the division factor is 2. We see that the duty cycle is not 50% but 25%. This can be useful to provide the design in “evaluation” mode. The operating frequency is twice lower, and so is the performance. Finally, Figure 3c shows the gated clock. In this mode, the design does not function at all.

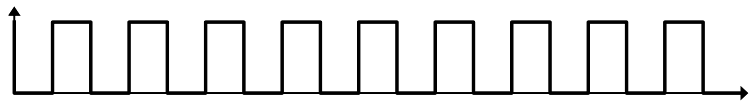

Fig. 3a Original clock

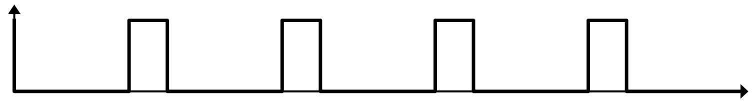

Fig. 3b Divided clock

Fig. 3c Gated clock

Acting on the clock has multiple advantages. First of all, it is a powerful way to completely disable the circuit. If the clock is not provided, most of the circuit's elements do not operate. Then, it requires very few additional logic resources. In the module presented in Figure 2, only one counter, one comparator, one multiplexer and one D flip-flop are used. It also allows to reduce the operating frequency, effectively getting the circuit to operate in evaluation mode.

The main drawback of such a scheme is that it inherently requires to alter the clock distribution network. This can be problematic in certain designs because the clock distribution network is usually very precisely tuned to meet timing requirements.

Therefore, next subsection proposes another way to act on the clock signal, by dynamically modifying the PLL configuration.

3.1.2 Dynamic phase-locked loop (PLL) reconfiguration

Another way to act on the clock signal is to directly deal with the phase-locked loop (PLL). In most of the integrated circuits, the clock signal is handled by a PLL. It allows to generate multiple clean clock signals, which can have a different frequency, to different parts of the circuit. It is then distributed by the clock tree.

In modern FPGAs, such as Altera Arria V, Cyclone V or Stratix V families [8] the PLL can be dynamically reconfigured. That is, the multiplication and division factors can be dynamically tuned. The PLL is then actually used as a frequency synthesiser.

The output frequency of the PLL is given by the following formula:

$$f_{out} = f_{in} \cdot \frac{M}{N.C} \tag{2}$$

C is the post-scale output counter. M is the feedback counter. N is the prescale counter. The values for M, N and C can be dynamically changed in order to tune the operating frequency.

In order to do this on FPGAs, a vendor-specific IP core must be instantiated. It is controlled by a dedicated finite-state machine (FSM), responsible for providing the C, M and N values. Once the parameters have been sent, the actual reconfiguration starts. The PLL unlocks, and then locks again on the new frequency. This is illustrated in Figure 4.

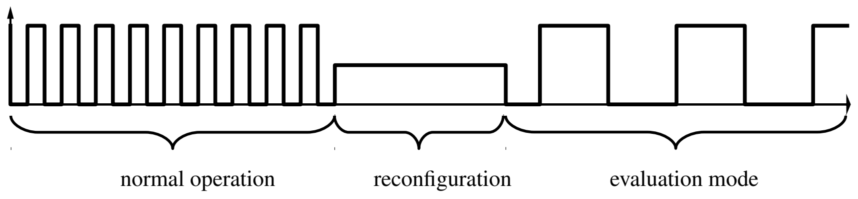

Fig. 4 Output clock during a reconfiguration

The normal mode of operation corresponds to the maximum frequency. Then the PLL is reconfigured, and locks again to the new frequency. This one is lower, and corresponds to the circuit operating in evaluation mode.

The main advantage of using the PLL as a frequency synthesiser is the flexibility it provides. Indeed, by individually setting the M, N and C parameters, it is possible to precisely tune the frequency. Moreover, since the reconfiguration process is natively supported by the PLL, the designer ensures that the output clock meets the specifications.

On the other hand, such reconfiguration feature might not be found in all the PLLs. For example, only recent versions of Altera's FPGAs support this feature. Another drawback is the area overhead. Indeed, instantiating the reconfiguration engine and the controlling FSM requires a lot of logic resources. This will be extensively discussed in Section 4.1.

3.2 Inputs/outputs

3.2.1 Embedded flip-flops

In most of the designs, the inputs are synchronised to be handled properly and avoid metastable states. The D flip-flops used to achieve this synchronisation often have an *enable* input. This *enable* input prevents new data to be sampled by the D flip-flop if it is driven low. By controlling this *enable* input, it is then possible to prevent the design from receiving new data from its inputs.

For example, for most of the FPGAs, the input/output blocks embed this type of D flip-flops. In order to specifically use this D flip-flop, some directives should be inserted in the design.

For Altera devices:

```
ATTRIBUTE useioff      : BOOLEAN;
ATTRIBUTE useioff OF e : SIGNAL IS true;
```

For Xilinx devices:

```
ATTRIBUTE IOB      : STRING;
ATTRIBUTE IOB OF e : SIGNAL IS "TRUE";
```

For Lattice devices:

```
USE DIN TRUE CELL "e";
USE DOUT TRUE CELL "e";
```

The advantage of such technique is to re-use existing elements of the design. By using flip-flops which are already implemented, the overhead is very limited. It also has a strong impact on design operation since it prevents new data to be loaded. However, it requires a specific type of flip-flop, since an *enable* input is necessary.

3.2.2 Fuses/anti-fuses

In 2014, Basak et al. also proposed to act on the inputs of a circuit to get it to operate properly or not [9]. They propose to integrate anti-fuses in the chip's pins. Those anti-fuses are blown or not according to an authentication key. If the wrong key is supplied, then the wrong fuses are blown and the device is not usable. After the fuses are blown, the correct inputs and outputs are accessible and the device operates normally.

An interesting feature here is that if a system integrator obtains an integrated circuit on which fuses are already blown, then it is obviously a refurbished one. Therefore, such scheme also helps in fighting other types of threats on design intellectual property.

3.3 Processor

3.3.1 Processor's program counter

Among complex systems, some integrate a soft-core processor in the FPGA fabric in order to execute programs. Such processor is described in a hardware description language and instantiated. Altera Nios II [10] and Xilinx MicroBlaze [11] are examples of proprietary soft-core processors. An example of open-source soft-core processor is the Plasma CPU, available on the IP cores repository Opencores [12]. Moreover, some SoC actually include a wired processor. For example, recent Altera Cyclone V SoCs integrate a dual-core ARM Cortex-A9 processor [1].

In order to disable a processor, acting on the program counter, also called instruction pointer, is a very effective solution. The program counter is a register that gives the address of the instruction being currently executed. Therefore, by controlling its value, it becomes possible to prevent new instructions from being executed. This can effectively halt the processor. Moreover, such halting can be set and released multiple times during the device's lifetime, allowing to achieve evaluation periods for instance. Therefore, acting on the program counter is a versatile way to *license* the device. In case of counterfeiting or overbuilding, it can also obviously be used to render the processor unusable by permanently forcing the program counter to a fixed value.

The detailed locking process is presented in Algorithm 1.

Algorithm 1: *Backup* and *locking* procedure

```
if locking request then
    if No branching or long instruction going on then
        if No branching or long instruction coming then
            $PC_{backup} \leftarrow PC$
Wait for ongoing instruction to finish then
$PC \leftarrow$ "000...000"
$instruction \leftarrow$ NOP
Locking completed
```

The first thing to do is to intercept the locking request. After that, it is important to verify that no problematic instruction is currently being executed. Indeed, if this is the case, then the return to normal operation is uncertain. Problematic instructions are long and branching instructions. Long instructions can not be stopped during their execution, and should terminate before. Similarly, during a branching instruction, the locking request should be postponed. This might cause the branching instruction to be skipped. The locking request should also be postponed if a problematic instructions is meant to be executed during the next clock cycle. In order

[1] https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.html

to detect these instructions, the opcodes corresponding to problematic instructions can be read directly from the memory bus. These opcodes are provided by the processor's designer. This decision is based on practical experiments. It is important to ensure that a correct backup of the processor's current state is possible. Locking requests are not time-critical and can be postponed for several clock cycles to ensure processing integrity.

After that, the current value of the program counter is stored in a dedicated register: PC_{backup}. At the end of the running instruction, which is not problematic, the program counter is set to a non-functional value. In Algorithm 1, we took the example of the zero value ("000...000") but this can be different depending on the processor. The instruction register is set to *NOP*. This is to avoid executing the same instruction over and over when the processor is locked. It could modify its internal state and make the return to normal operation impossible.

Finally, the locking process is considered as completed.

In order to return to normal operation, the previous program counter value should be restored. This is shown in Algorithm 2.

Algorithm 2: *Restore* procedure

```
if unlocking request then
    $PC \leftarrow PC_{backup}$
Wait for instruction to be loaded then
Unlocking completed
```

An interesting feature of this locking scheme is that it is fully reversible. Indeed, if the locking procedure has been followed properly, the instruction during which the locking occurred is not problematic. Therefore, the system can then be unlocked and start again without problem.

3.4 Buses

Buses integrity is crucial for correct communication between the different components of a system. Integrity can be more precisely defined in two terms: value and position. Thus data from a bus is sound if it has a correct value and it is correctly ordered.

Therefore, by acting on either the value or the position of the bus data, we can alter the bus operation. The first option is then to scramble the bus lines. The second option is to randomly mask the bus data.

For the subsequent sections 3.4.1 and 3.4.2 we assume that the bus is error-free. The input value is identical to the output value during normal operation.

3.4.1 Deterministic scrambling

A bus can be defined as the following function f:

$$f : \{0,1\}^n \rightarrow \{0,1\}^n \tag{3}$$

$$\forall x \in \{0,1\}^n : f(x) = x \tag{4}$$

It can thus be referred to as the identity function.

We define a deterministic scrambling function σ as :

$$\sigma : \{0,1\}^n \rightarrow \{0,1\}^m \; with \; n \geq m \tag{5}$$

$$\forall x \in \{0,1\}^n : \sigma(x) \neq x \tag{6}$$

However, such function can be heavy to implement. The requirement given in equation (6) is hard to fulfil for all x.

Therefore, we can defined a relaxed version of the deterministic scrambling function σ_R as :

$$\sigma_R : \{0,1\}^n \rightarrow \{0,1\}^m \; with \; n \geq m \tag{7}$$

such that for *most of* the input values:

$$\sigma_R(x) \neq x \tag{8}$$

In fact, the relaxed version is sufficient for the usage we consider here. Indeed, disturbing a bus for even half of the input values is enough to render the overall system unusable.

The other point is to make the scrambling controllable by an additional input such that the scrambler can be turned on and off. A simple 2-to-1 n-bit multiplexer can be used to this end, selecting between the original bus data and the scrambled one. This is shown on Figure 5.

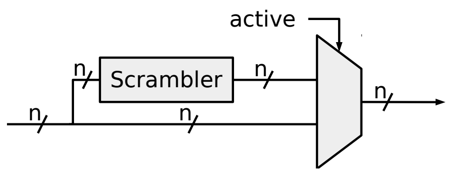

Fig. 5 Integration of the scrambler on an n-bit bus

From a practical point of view, implementing a scrambler is trivial. An n-bit circular shifter defined as $\sigma_R(x_i) = x_{i-1 \; mod \; n}$ and shown in Figure 6 is efficient. It is only a relaxed deterministic scrambler since it does not alter the data if it consists or only 0s or only 1s.

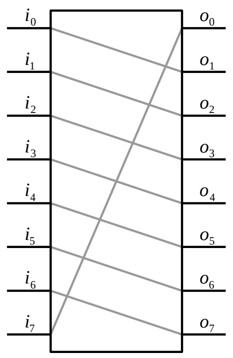

Fig. 6 8-bit circular shifter

However, there are even more trivial structures which can be used to scramble the bus. We do not give more details here as the chosen method is strongly dependent on the bus purpose.

Implementation details can require to add extra specifications to the scheme shown in Figure 5. As shown on Figure 1 of this chapter, a scrambler can be added to the address bus of the shared memory. This is a suitable choice, since reading from a wrong memory address disturbs the system heavily. However, writing to an unauthorised memory address could potentially alter the ability of the system to recover once the scrambler will be deactivated. For instance, the program memory could be irremediably altered. Therefore, in this case, the bus should be scrambled only during read operations, not write.

3.4.2 Pseudo-random masking

Another way to corrupt a bus is to act on the actual data which is transmitted through it. To this end, pseudo-random masking can be used.

In order to get pseudo-randomness, we use a linear feedback shift register, or LFSR. Then, the shift register state bits are XOR-ed bitwise with the bus lines. For an n-bit bus, an n-bit shift register is used. If the feedback polynomial is carefully chosen, i.e. is primitive, a $2^n - 1$ clock-cycles period can be obtained. This is shown in Figure 7.

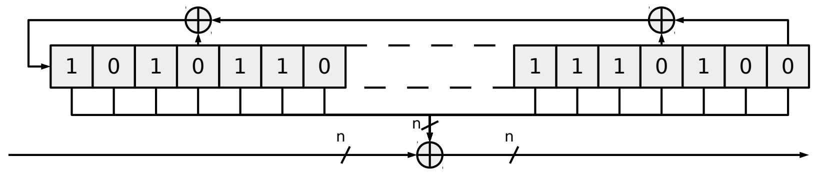

Fig. 7 Pseudo-random masking of a bus using an LFSR

Similarly, the note made in the previous section about implementation-specific issues also applies to the pseudo-random masking scheme.

In order to reduce the power overhead induced by the LFSR, it can be clocked at a lower frequency then the nominal one of the design. Indeed, power consumption is proportional to the operating frequency.

3.5 Finite state machine

The first way to modify the FSM is to add extra states before the original reset state. This is described in Section 3.5.1. The second option is to duplicate intermediate states to stop normal operation if the correct key is not provided. This is detailed in Section 3.5.2.

3.5.1 Pre-reset states

The first possibility to modify the FSM is to add extra states before the original reset state [4, 13, 14]. This way, when the system is powered up or reset, it starts again in these extra states. In order to reach normal operation, the design must transition from one extra state to the other until it reaches the original reset state. If the state transitions of the extra states are only known to the original designer, then an attacker will not be able to reach the original reset state. The only possibility would be to explore all the extra states until the original reset state is reached.

The extra states can come at no cost if the original state machine is encoded in a way that so-called *don't care* states exist. If the FSM's states follow binary encoding, then an M-state FSM must use at least $\lceil log_2(M) \rceil$ D flip-flops to store the current state's value. If the number of flip-flops used is n, then there are $2^n - M$ states which are not used. These are *don't care* states. They can be used to encode the extra states.

A graphical representation of the modified FSM is shown in Figure 8.

In this example, the original FSM includes five states. $\lceil log_2(5) \rceil = 3$, so three flip-flops are needed for state encoding. However, three flip-flops can encode $2^3 = 8$ states. Therefore, the three *don't care* states can be used as pre-reset states.

The new reset state is $S'0$. In order to transition to the original reset state $S0$, the correct values for K_0, K_1 and K_2 should be sent. If one key bit is wrong, then $S'0$ is reached again.

The advantage of such technique is to have low overhead since it makes use of *don't care* states. It has several drawbacks though. First of all, from a security point of view, this locking scheme exhibits a key even though it is not secure on its own. This can be misleading and get the designer to consider the scheme secure. However, security should rely on a cryptographic primitive. Second of all, transitions from one state to the other could be detected by the transient power consumption of switching flip-flops which encode the current state. Thus, finding the right key becomes trivial.

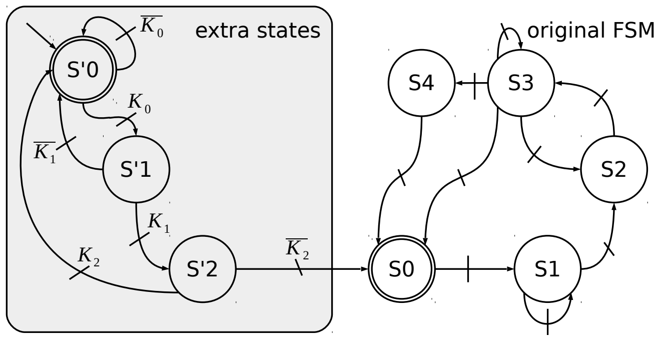

Fig. 8 Pre-reset states with key $(K_0K_1K_2) = 110$

One option explored to make the scheme more secure is to initialise the state flip-flops to a random value, given by the response of a PUF to a specific challenge [4]. Since only the designer knows the PUF's challenges/responses pairs, only he can find out the start-up state associated with a challenge. Therefore, in order to reach the original FSM, he must provide the system integrator with the right sequence of inputs to provide to the FSM. However, this only makes each FSM instance behave differently. It does not account for the two drawbacks previously described. Furthermore, it does not consider the variability of PUF's responses. If the PUF's response differs, the start-up state expected by the designer is different than the actual one of the powered-up device. Thus the designer can not provide the appropriate sequence of inputs to unlock the circuit.

Another option to modify the FSM is to duplicate specific states. This is described in the following section.

3.5.2 Duplicated states

Similarly, it is possible to use *don't care* states to duplicate some intermediate states. This is described in [15], and shown in Figure 9. In this example, state $S2$ is duplicated.

The transitions from $S1$ to one of the duplicated states $S21$, $S22$, $S23$ and $S24$ is controlled by the output of a PUF, called random unique block in [15]. After that, in order to transition to the next state, here $S3$, a specific key must be applied to the FSM's inputs. This key is associated with the PUF's response, and known only by the designer. If the wrong key is applied, the FSM does not transition to the next state and remains locked.

The advantages and drawbacks of this method are the same as the ones describes in Section 3.5.1.

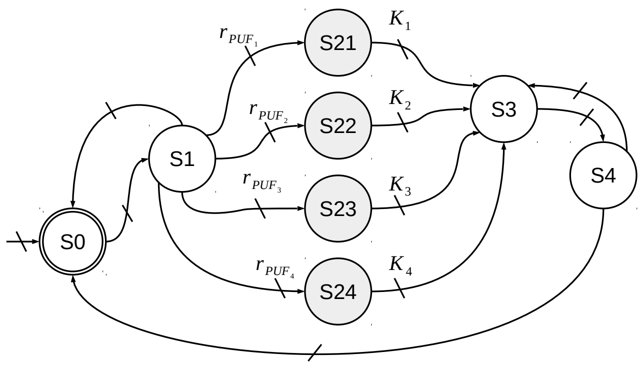

Fig. 9 Duplicated states S21, S22, S23 and S24

4 Implementation on FPGA and results

We implemented the on-chip locks proposed in Section 3 on FPGA. We first give the cost for all solutions in terms of hardware resources. We then them implement on reference designs to estimate the implementation overhead. All the results are given with optimisation for lowest area. We used Quartus II 13.1 and ISE 13.4 for synthesis.

4.1 Hardware resources

The experimental results obtained are given in Table 2. The implementation was carried out on two FPGA families: Altera Cyclone III and Xilinx Spartan 3. They are provided as the number of 4-input look-up tables (LUTs) and D flip-flops used for the implementation.

First, we can see that logic resources usage is very low for all the locks, except for the PLL reconfiguration. Indeed, in this case, the heaviest module is the one responsible for achieving the reconfiguration. It is provided by the FPGA manufacturer and can hardly be modified or optimised.

Conversely, all the other locks require very few logic resources. The most lightweight one consists in acting on the *enable* input of the input/output flip-flops. For buses, we give the required resources in terms of the bus-width. They are always proportional to the bus width. When extra states are added to the FSM, either as pre-reset or duplicated states, the resources overhead grows logarithmically with respect to their number. We didn't take into account here the possibility to reuse *don't care* states. This would reduce the required resources even further.

Modifying the program counter is a specific process for each processor. This is detailed in Section 4.2.2 for the Plasma CPU.

Table 2 Implementation results of on-chip locks alone on Altera Cyclone III and Xilinx Spartan 3

Modified feature	On-chip lock	#4-input LUTs	#D flip-flops
Clock circuitry	Reconfigurable PLL[a] (+ control FSM)	247(+55)	118(+18)
	Clock-gating module	9	6
Inputs/outputs	Inputs/outputs DFF enable	0	0
Interconnection bus	Deterministic scrambling[b]	8n/5	0
	Pseudo-random masking[b]	n	n
FSM	Pre-reset states[c]	log(n)	log(n)
	Duplicated states[c]	log(n)	log(n)

[a]only on Cyclone III, not available on Spartan 3
[b]of an *n*-bit bus
[c]for *n* extra states

4.2 Reference designs

We then propose to implement the locks on two reference designs: an Ethernet controller and a soft-core processor. Both designs are available on the Opencores website [12]. For comparison, we give the resources required to implement the original designs in Table 3.

Table 3 Required resources to implement the original designs

Unmodified	Altera Cyclone III		Xilinx Spartan 3	
design	#4-input LUTs	#D flip-flops	#4-input LUTs	#D flip-flops
Ethernet controller	275	108	357	99
Plasma CPU	2395	452	2901	394

4.2.1 Ethernet controller

The first reference design is an Ethernet controller. It is a fairly small design, mainly consisting in an FSM.

We first modified the clock circuitry. On the one hand, the clock gating module requires 3% more combinational fabric and 6% more D flip-flops. The resources overhead is then rather low. On the other hand, implementing a reconfigurable PLL doubles the required resources, and is clearly not practical.

As expected, acting on the D flip-flop *enable* input requires no extra-resources.

When acting on the bus, scrambling is cheaper than masking. We implemented them both on a 32-bit bus. Scrambling requires extra combinational logic, around 9% more. On the other hand, pseudo-random masking needs D flip-flops to implement the LFSR. Therefore, the associated overhead is quite high, around 30% more D flip-flops.

Finally, we also modified the FSM in both ways. First, we added 32 pre-reset states. Even if it only requires one extra flip-flop, the combinational logic handling the transitions between the extra states is heavy, and requires almost 25% more resources. Then, we duplicated one of the state 32 times. Similarly, only one extra flip-flop was added to the design but the requirement for combinational logic exceeds 70% here, which is excessive.

All the results of the implementation on the Ethernet controller are shown in Table 4.

Table 4 Implementation of on-chip locks on the Ethernet controller

Modified	Altera Cyclone III		Xilinx Spartan 3	
design	#4-input LUTs	#D flip-flops	#4-input LUTs	#D flip-flops
Clock-gating module	284 (+3%)	114 (+6%)	367 (+3%)	105 (+6%)
PLL reconfiguration	522 (+90%)	226 (+109%)	*Not available*	
Inputs/outputs DFF enable	275 (+0%)	108 (+0%)	357 (+0%)	99 (+0%)
Deterministic scrambling[a]	297 (+8%)	108 (+0%)	388 (+9%)	99 (+0%)
Pseudo-random masking[a]	313 (+14%)	140 (+30%)	391 (+9%)	131 (+32%)
32 pre-reset states	343 (+25%)	109 (+1%)	425 (+19%)	100 (+1%)
Duplicated state (x32)	493 (+80%)	109 (+1%)	612 (+71%)	100 (+1%)

[a]for a 32-bit bus

For a design of this size, we can then estimate that only input/output locking is suitable. Adding a clock-gating module can be also considered, since the overhead is still rather low. All the other modifications lead to an important overhead.

4.2.2 Plasma CPU

A larger design is now presented, the soft-core processor Plasma CPU.

Here, the clock gating module is even cheaper in terms of resources. However, PLL reconfiguration remains expensive, with 13% more combinational resources and 30% more D flip-flops.

Like before, using integrated input/output D flip-flops adds no logic resources.

Modifying the bus becomes affordable with this kind of large designs. Scrambling it in a deterministic way induces almost no overhead. Pseudo-random masking leads to low overhead, below 8% Therefore, it can be implemented as a powerful way to disturb the bus data.

Since the Plasma CPU does not comprise an FSM, pre-reset and duplicated states could not be implemented on this design.

Finally, being able to control the program counter value is also quite expensive, and requires 10% extra resources. However, this was implemented on an already existing design. We assume it could be implemented in a more lightweight way if this feature was taken into account during the design phase.

All the implementation results are provided in Table 5.

Table 5 Implementation of on-chip locks on the Plasma CPU

Modified	Altera Cyclone III		Xilinx Spartan 3	
design	#4-input LUTs	#D flip-flops	#4-input LUTs	#D flip-flops
Clock-gating module	2399 (+0.17%)	458 (+1%)	2932 (+1%)	400 (+2%)
PLL reconfiguration	2697 (+13%)	588 (+30%)	*Not available*	
Inputs/outputs DFF enable	2395 (+0%)	452 (+0%)	2901 (+0%)	394 (+0%)
Deterministic scrambling[a]	2430 (+1%)	452 (+0%)	2894 (+0%)	394 (+0%)
Pseudo-random masking[a]	2446 (+2%)	484 (+7%)	2927 (+1%)	426 (+8%)
Program counter halt	*Not implemented*		3186 (+10%)	428 (+9%)

[a]for a 32-bit bus

On larger designs, implementing more complex locks is possible. The associated overhead is limited, and even multiple locks could be integrated.

5 Discussion: partial locking

For some of the features presented previously, it is possible to achieve partial locking. Partial locking can refer to the following features:

- Lower performance: lower operating frequency...
- Less features: reduced instruction set...
- Limlited period of use: trial period.

However, the system is still perfectly functional.

Such partial locking feature is very interesting from the point of view of the designer. In fact, it allows the designer to provide the design in evaluation mode. This way, the design can be thoroughly tested by the future user before buying it. With the increasing number of designs provided as IP cores, this allows to support a business model similar to the one used in for software distribution. The product is first provided as an evaluation version, and can then be fully unlocked to perform optimally. This offers interesting flexibility to the licensing model.

What is important to partially lock a design is to choose a locking scheme that provides some granularity. Among the modifications we have shown in the previous sections, acting on the clock circuitry is the only one able to achieve this. Indeed, as said before, the clock frequency is directly related to the performance of the design. Therefore, acting on the clock frequency using a clock gating module or by dynamically reconfiguring the PLL is a way to tune system performance.

In order to implement this kind of functionality, a controller should also be implemented to handle the different states in which the system can operate: locked, evaluation or unlocked. Different commands and the associated unique keys are controlling the transitions from one state to another. It leads to additional resources overhead, which must be taken into account.

6 Conclusion

This chapter shows how some features, already existing in most electronic designs, can be turned into powerful on-chip locks. Since they are already present, using them leads to low resources overhead. We provide different techniques about how to effectively implement these locks. We also give details on partial locking. This can help to achieve a more flexible licensing model for IP cores, allowing to provide devices in evaluation mode.

The most suited way to modify a design to make it lockable seems to be the modification of the clock circuitry. It is lightweight and can effectively tune the performance of the design.

References

1. Frontier-Economics, "Estimating the global economic and social impacts of counterfeiting and piracy," Business Action to Stop Counterfeiting and Piracy (BASCAP), Tech. Rep., 2011.
2. C. Gorman, "Counterfeit chips on the rise," *IEEE Spectrum*, vol. 49, no. 6, pp. 16–17, 2012.
3. U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.
4. Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," in *USENIX Security*, Boston MA, USA, August 2007, pp. 291–306.
5. R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.
6. L. Bossuet and D. Hély, "SALWARE: Salutary hardware to design trusted IC," in *Workshop on Trustworthy Manufacturing and Utilization of Secure Devices, TRUDEVICE*, 2013.
7. B. Colombier and L. Bossuet, "Functional locking modules for design protection of intellectual property cores," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, Vancouver, Canada, May 2015, p. 233.
8. Altera. (2015, December) Implementing fractional pll reconfiguration with altera pll and altera pll reconfig ip cores.
9. A. Basak, Y. Zheng, and S. Bhunia, "Active defense against counterfeiting attacks through robust antifuse-based on-chip locks," in *IEEE 32nd VLSI Test Symposium*, Napa CA, USA, April 2014, pp. 1–6.
10. Altera. Nios. [Online]. Available: https://www.altera.com/products/processors/overview.html
11. Xilinx. Microblaze. [Online]. Available: http://www.xilinx.com/products/intellectual-property/microblazecore.html
12. Opencores. [Online]. Available: opencores.org

13. R. S. Chakraborty and S. Bhunia, “Security against hardware trojan through a novel application of design obfuscation,” in *International Conference on Computer-Aided Design*. ACM, 2009, pp. 113–116.
14. ——, “HARPOON: an obfuscation-based soc design methodology for hardware protection,” *IEEE Transations on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.
15. Y. Alkabani, F. Koushanfar, and M. Potkonjak, “Remote activation of ICs for piracy prevention and digital right management,” in *IEEE/ACM international conference on Computer-aided design*, Beijing, China, October 2007, pp. 674–677.