Universidade Estadual de Campinas
Instituto de Computação

Luciana Bulgarelli Carvalho

LEON3-CONF: processador LEON3 com confidencialidade de instruções e de dados

CAMPINAS
2018

Luciana Bulgarelli Carvalho

LEON3-CONF: processador LEON3 com confidencialidade de instruções e de dados

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestra em Ciência da Computação.

Orientador: Prof. Dr. Guido Costa Souza de Araújo

Este exemplar corresponde à versão final da Dissertação defendida por Luciana Bulgarelli Carvalho e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2018

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.
ORCID: https://orcid.org/0000-0002-3510-8914

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C253L

Carvalho, Luciana Bulgarelli, 1977-
LEON3-CONF : processador LEON3 com confidencialidade de instruções e de dados / Luciana Bulgarelli Carvalho. – Campinas, SP : [s.n.], 2018.

Orientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Computadores - Medidas de segurança. 3. Criptografia de dados (Computação). 4. FPGA (Field Programmable Gate Array) – Projetos e construção. I. Araújo, Guido Costa Souza de, 1962-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: LEON3-CONF : LEON3 processor with instructions and data confidentiality
Palavras-chave em inglês:
Computer architecture
Computer security
Data encryption (Computer science)
Field programmable gate arrays - Design and construction
Área de concentração: Ciência da Computação
Titulação: Mestra em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Araújo, Guido Costa Souza de
Ivan Saraiva Silva
Mario Lúcio Côrtes
Data de defesa: 18-12-2018
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Luciana Bulgarelli Carvalho

LEON3-CONF: processador LEON3 com confidencialidade de instruções e de dados

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo
  IC/UNICAMP
- Prof. Dr. Ivan Saraiva Silva
  CCN/UFPI
- Prof. Dr. Mario Lúcio Côrtes
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 18 de dezembro de 2018

Dedicatória

Dedico este trabalho aos meus amados pais, José Luiz e Suzana, ao meu querido irmão, Leandro, à minha querida cunhada, Eliene, e aos meus amados e eternos sobrinhos: Helena e Pedro (*in memory*).

Agradecimentos

Agradeço a Deus pela vida e por mais um sonho realizado, ao professor Dr. Guido pela oportunidade, confiança, compreensão e por toda a ajuda técnica, aos meus pais, José Luiz e Suzana, pelo apoio, amor, paciência e dedicação.

Agradeço também a todos que ajudaram (tecnicamente ou não, diretamente ou não) na realização deste trabalho.

Resumo

Garantir a privacidade e integridade de dados e instruções e a corretude da computação que se deseja realizar é o objetivo central em segurança computacional moderna. Mecanimos que evitem ataques aos dados e às instruções durante a execução de programas devem garantir a sua confidencialidade desde o momento em que eles são armazenados na memória até quando são executados internamente dentro do processador. O processador LEON3-CONF, proposto neste trabalho, garante a confidencialidade de instruções e/ou de dados na memória externa e durante seu transporte através do barramento processador-memória. Para isto, o LEON3-CONF encripta as instruções e/ou os dados antes de enviá-los à região não segura (composta pelo barramento processador-memória e pela memória externa) e os decripta antes de processá-los e/ou armazená-los. A arquitetura da camada de segurança do LEON3-CONF foi projetada para permitir que um algoritmo de encriptação e decriptação seja adicionado ao processador de forma simples e visando minimizar o impacto no tempo de execução dos programas. O algoritmo de encriptação e decriptação implementado nesta versão do LEON3-CONF é o AES (*Advanced Encryption Standard*).

O objetivo principal deste trabalho é desenvolver e avaliar o impacto de cifração da memória e do barramento na arquitetura LEON3 e não propor um algoritmo novo para esta tarefa. A avaliação do desempenho dos mecanimos de cifração do LEON3-CONF foi feita em uma plataforma FPGA usando um conjunto de programas em linguagem C. Os resultados experimentais revelaram que: (a) o tempo de execução de um programa com apenas dados seguros é, no pior caso, 2,11 vezes o tempo de execução do mesmo programa sem instruções e dados seguros; (b) o tempo de execução de um programa com apenas instruções seguras é, no pior caso, 1,86 vezes o tempo de execução do mesmo programa sem instruções e dados seguros; e (c) o tempo de execução de um programa com instruções e dados seguros é, no pior caso, 2,66 vezes o tempo de execução do mesmo programa sem instruções e dados seguros. O resultado da síntese do LEON3-CONF indica que a camada de segurança utiliza 36,7% da lógica combinacional e 15,8% dos registradores do LEON3-CONF e que o algoritmo de encriptação e decriptação AES utiliza 84,9% da lógica combinacional e 41,9% dos registradores da camada de segurança.

Abstract

Ensuring the privacy and integrity of data and instructions and the correctness of the computing that you want to accomplish is the central goal in modern computational security. Mechanisms that prevent attacks on data and instructions while running programs must guarantee their confidentiality from the time they are stored in memory until when they are run internally within the processor. The LEON3-CONF processor, proposed in this work, guarantees the confidentiality of instructions and/or data in the external memory and during their transport through the processor-memory bus. To do this, LEON3-CONF encrypts the instructions and/or data before sending them to the non-secure region (composed of the processor-memory bus and external memory) and decrypts them before processing and/or storing them. The architecture of the LEON3-CONF security layer was designed to allow an encryption and decryption algorithm to be added to the processor in a simple way and to minimize the impact on the programs runtime. The encryption and decryption algorithm implemented in this version of LEON3-CONF is AES (Advanced Encryption Standard).

The main objective of this work is to develop and evaluate the impact of memory and bus encryption on the LEON3 architecture and not propose a new algorithm for this task. The performance evaluation of the LEON3-CONF encryption mechanisms was done in an FPGA platform using a set of C language programs. The experimental results showed that: (a) the runtime of a program with only safe data is, in the worst case, 2.11 times the runtime of the same program without safe instructions and data; (b) the runtime of a program with only safe instructions is, in the worst case, 1.86 times the runtime of the same program without safe instructions and data; and (c) the runtime of a program with safe instructions and data is, in the worst case, 2.66 times the runtime of the same program without safe instructions and data. The result of the LEON3-CONF synthesis indicates that the security layer uses 36.7% of the LEON3-CONF combinational logic and 15.8% of the LEON3-CONF registers and that the AES encryption and decryption algorithm uses 84.9% of the security layer combinational logic and 41.9% of the security layer registers.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A segurança computacional é um importante requisito de diversos sistemas computacionais e a sua demanda é crescente. Entre estes sistemas computacionais destacam-se: sistemas embarcados (*embedded systems*), dispositivos móveis (*mobile devices*), *Internet of Things* (IoT)[27, 28] e *Cloud Computing*[35].

De acordo com *National Institute of Standards and Technology* (NIST)[26, 36], a segurança computacional é a proteção de um sistema de informação com o objetivo de garantir a integridade, a disponibilidade e a confidencialidade dos recursos deste sistema. Estes recursos incluem hardware, software, firmware, informações/dados e telecomunicações.

A integridade de um sistema de informação inclui a integridade dos dados e a integridade do sistema. A integridade dos dados garante que os programas e as informações sejam alterados apenas de maneira específica e autorizada. Por outro lado, a integridade do sistema garante que o sistema realize a sua função de maneira intacta, isto é, livre de manipulações não autorizadas. Já a disponibilidade de um sistema de informação garante que os recursos deste sistema funcionem prontamente e que o serviço não seja negado aos usuários autorizados. A confidencialidade garante que as informações privadas ou confidenciais não sejam divulgadas a indivíduos não autorizados.

Um sistema computacional básico é composto por um processador conectado a uma memória externa através do barramento processador-memória (Figura 1.1). O processador pode ser considerado a região segura do sistema, pois as instruções e os dados têm sua confidencialidade garantida no interior do encapsulamento do circuito integrado, e o custo para um atacante extrair as informações de dentro do *die* do processador, que está interno ao encapsulamento, é imensamente maior do que fazer isto externamente. Deste modo, o barramento processador-memória e a memória externa compõem a região não segura

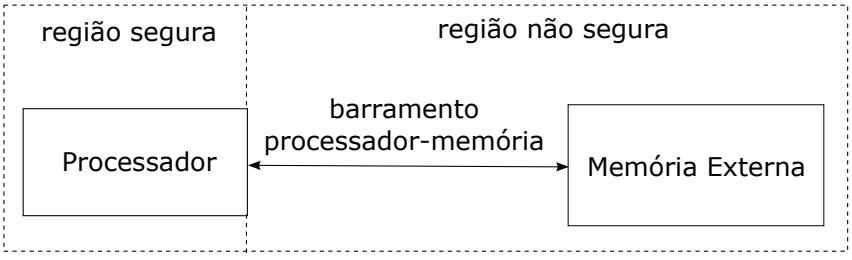

Figura 1.1: Sistema dividido em região segura e não segura.

do sistema, pois as instruções e os dados no barramento processador-memória e/ou na memória externa podem ser acessados (lidos ou modificados) com facilidade por indivíduos não autorizados.

Para garantir a confidencialidade das instruções e dos dados protegidos nesta região não segura, estes são encriptados pelo processador antes de serem enviados à região não segura. E, quando o processador os recebe da região não segura, primeiro, o processador os decripta e, depois, os processa e/ou os armazena em sua(s) cache(s). Desta forma, as instruções e os dados protegidos estão encriptados enquanto estiverem na região não segura. Esta estratégia para garantir a confidencialidade é amplamente adotada na literatura e na indústria (vide Capítulo 4) e será utilizada também neste projeto.

O objetivo do LEON3-CONF (processador LEON3 com confidencialidade de instruções e de dados) desenvolvido neste projeto é garantir a confidencialidade das instruções e dos dados protegidos. Assim, a camada de segurança (Seção 3.1) foi projetada, e a arquitetura do processador LEON3[5] foi modificada para que a camada de segurança fosse conectada ao barramento AMBA AHB *slave* zero. O algoritmo de encriptação e decriptação selecionado e implementado na camada de segurança foi o AES (*Advanced Encryption Standard*)[21].

A verificação da camada de segurança e do LEON3-CONF foi realizada através da simulação do LEON3-CONF usando um conjunto de programas desenvolvidos em Assembly e em linguagem C. Estes programas também foram executados no LEON3-CONF programado em FPGA.

Com o intuito de quantificar o desempenho do LEON3-CONF com o algoritmo de encriptação e decriptação escolhido, outro conjunto de programas em linguagem C foi desenvolvido. Cada um destes programas foi executado no LEON3-CONF (programado em FPGA) com quatro configurações diferentes: instruções e dados não seguros, instruções não seguras e dados seguros, instruções seguras e dados não seguros, instruções e dados seguros. O tempo de execução (número de períodos de clock) foi medido para cada uma destas configurações (Tabela 5.11).

1.1 Motivação

A motivação deste projeto é propor uma arquitetura que garanta a confidencialidade de instruções e dados seguros no barramento processador-memória e na memória externa (Figura 1.1) e quantificar o desempenho desta arquitetura.

A estratégia adotada para assegurar a confidencialidade das instruções e dos dados seguros consiste em garantir que as instruções e os dados seguros estarão na forma clara apenas na região segura e encriptados na região não segura. Para tanto, o processador encripta as instruções e os dados seguros antes de enviá-los à memória externa e os decripta quando os recebe da memória externa e antes de processá-los e/ou armazená-los em sua(s) cache(s).

Esta estratégia é utilizada por nove dos dez artigos analisados para este projeto (Capítulo 4). A exceção é o artigo *Design of A Pre-Scheduled Data Bus for Advanced Encryption Standard Encrypted System-on-Chips*[48] que utiliza uma arquitetura diferente: os dados

seguros são armazenados na forma clara e em uma memória interna acessada apenas pelo DMA (Figura 4.2).

Destes nove artigos, seis ([11, 20, 39, 45, 46, 49]) utilizam o algoritmo de encriptação e decriptação AES em *counter mode* e três ([15, 16, 43]) o AES em CBC *(Cipher Block Chaining) mode*. Estes nove artigos apresentam as vantagens, as desvantagens e os desempenhos de suas arquiteturas de acordo com o algortimo de encriptação e decriptação escolhido (AES em *counter mode* ou em CBC *mode*). Além disso, todos arfirmam que o uso do algoritmo AES é impraticável devido ao desempenho deste algoritmo. Entretanto, nenhum dos artigos quantifica o desempenho da arquitetura quando o algoritmo AES é adotado.

O algoritmo de encriptação e decriptação escolhido para este projeto foi o AES devido à sua qualidade em garantir a confidencialidade das instruções e dos dados encriptados. Além disso, este projeto contribui com a análise quantitativa de desempenho da arquitetura considerada (Figura 1.1) quando o algoritmo AES é utilizado.

1.2 Objetivos

Conforme dito anteriormente, o objetivo principal do projeto do LEON3-CONF é garantir a confidencialidade de instruções e de dados na memória externa e no barramento processador-memória. Com o intuito de alcançar este objetivo principal, o projeto do LEON3-CONF foi dividido nas seguintes tarefas:

1. Definir uma arquitetura que:
    - Encripte as instruções e os dados seguros antes do processador enviá-los à memória externa.
    - Decripte as instruções e os dados seguros recebidos da memória externa e antes do processador utilizá-los.
    - Permita a inclusão de algoritmos de encriptação e decriptação de uma maneira simples. Assim, futuras comparações de desempenho do LEON3-CONF com diferentes algoritmos poderão ser realizadas facilmente.
    - Divida a memória de instruções em um segmento para instruções seguras e outro para instruções não seguras.
    - Divida a memória de dados em um segmento para dados seguros e outro para dados não seguros.
2. Implementar a arquitetura definida em RTL (*Register Transfer Level*) sintetizável internamente à arquitetura do processador LEON3[5].
3. Implementar em RTL sintetizável o algoritmo de encriptação e decriptação AES[21] (*Advanced Encryption Standard*).
4. Utilizar apenas ferramentas sob licença GPL[29] (*GNU General Public License*) ou sob licença acadêmica para o desenvolvimento, a verificação e a análise quantitativa de desempenho do LEON3-CONF.

5. Verificar a funcionalidade do LEON3-CONF:

   (a) Desenvolver um conjunto de programas em Assembly e em linguagem C para verificar o LEON3-CONF;

   (b) Executar este conjunto de programas no LEON3-CONF através de sua simulação;

   (c) Executar este conjunto de programas no LEON3-CONF programado em FPGA (*Field Programmable Gate Arrays*).

6. Desenvolver um ambiente com programas e scripts para o desenvolvimento e a verificação do LEON3-CONF e a sua programação em FPGA.

7. Desenvolver um conjunto de programas em linguagem C para quantificar o desempenho do LEON3-CONF.

8. Quantificar o tempo de execução (número de períodos de clock) dos programas do item anterior executados no LEON3-CONF programado em FPGA. Cada um destes programas será executado com as seguintes configurações:

   - Instruções e dados não seguros;
   - Instruções não seguras e dados seguros;
   - Instruções seguras e dados não seguros;
   - Instruções e dados seguros.

1.3 Contribuições

As contribuições deste projeto incluem: o projeto do LEON3-CONF; a análise quantitativa de desempenho do LEON3-CONF com o algoritmo AES; scripts desenvolvidos para simular o LEON3-CONF e sintetizá-lo; programas em Assembly e em linguagem C desenvolvidos para verificar LEON3-CONF, um programa (em linguagem C) para encriptar o programa a ser executado no LEON3-CONF e programas (em linguagem C) para quantificar o desempenho do LEON3-CONF.

Este ambiente de desenvolvimento (composto por scripts e programas) pode ser reutilizado quando um outro algoritmo de encriptação e decriptação for adicionado à camada de segurança.

Esta dissertação está dividida da seguinte forma: o Capítulo 2 descreve o processador LEON3 e o algoritmo AES de encriptação e decriptação; o Capítulo 3 apresenta o processador LEON3-CONF, a sua arquitetura, o desenvolvimento do seu RTL, a sua verificação e a sua síntese; o Capítulo 4 lista os trabalhos já realizados nesta área, fazendo uma análise comparativa com o LEON3-CONF; o Capítulo 5 apresenta os resultados experimentais e faz uma análise da área e do desempenho do LEON3-CONF quando programado em FPGA. Finalmente, o Capítulo 6 conclui o trabalho.

Capítulo 2

Background

Este capítulo contém a descrição de conceitos importantes utilizados ao longo do desenvolvimento deste projeto. O capítulo está dividido em duas seções. A Seção 2.1 apresenta o processador LEON3, sua arquitetura e suas principais características. A Seção 2.2 detalha o algoritmo de encriptação e decriptação escolhido para este projeto: o algoritmo AES.

2.1 Processador LEON3

Esta seção apresenta o processador escolhido para este projeto: o processador LEON3. A seção está estruturada em duas divisões. A Seção 2.1.1 lista as principais características do processador LEON3 e as razões que motivaram a sua escolha. A Seção 2.1.2, por sua vez, detalha a arquitetura do *testbench* do LEON3 e a do processador LEON3 até o nível de abstração adequado para definir o limite entre a região segura e a não segura.

2.1.1 Visão Geral

O processador selecionado para este projeto foi o processador LEON3[5, 8, 7, 10]. A versão do código fonte do LEON3 utilizada neste trabalho foi a *grlib-gpl-1.1.0-b4108*[3].

As principais razões que motivaram a escolha do LEON3 foram:

1. O código fonte do LEON3:
   - é *open-source* sob licença GPL[29] (*GNU General Public License*);
   - está descrito em RTL (*Register Transfer Level*);
   - é sintetizável para FPGA (*Field Programmable Gate Arrays*) e para ASIC (*Application Specific Integrated Circuits*).
2. O LEON3 pode ser programado na placa Altera DE2-115 (Apêndice A.5) e pode ser acessado pelo *General Debug Monitor* GRMON2 (Apêndice A.6).

O LEON3 é um processador de 32 bits compatível com a arquitetura SPARC V8[31] e projetado para aplicações embarcadas. Outras características do LEON3 são:

- *Pipeline* com 7 estágios.
- Arquitetura Harvard.
- Cache de instruções e de dados separadas.
- Caches configuráveis.
- Unidade de Gerenciamento de Memória SRMMU (*SPARC V8 Reference Memory Management Unit*).
- Multiplicador e divisor implementados em hardware.
- Suporte para *debug on-chip*.
- Barramentos AMBA (*Advanced Microcontroller Bus Architecture*) 2.0 AHB (*Advanced High-performance Bus*) e APB (*Advanced Peripheral Bus*)[34].
- Suporte para multiprocessamento.
- Está descrito em VHDL - *VHSIC* (*Very High Speed Integrated Circuit*) *Hardware Description Language*.

O processador LEON3 possui uma cache de instruções e outra de dados, um controlador de cache para a cache de instruções e outro controlador para a cache de dados. As caches e os controladores de cache do LEON3 são configuráveis de acordo com:

- a associatividade (número de *ways*): *direct-mapped* (*single-way*) cache ou cache com 2, 3 ou 4 *ways* (*multiway*);
- o tamanho de cada *way*: 1 a 256 KiB (um KiB é igual a 1024 bytes);
- o tamanho de cada linha da cache: 16 ou 32 bytes;
- a política de substituição (*replacement policy*): LRU (*least-recently-used*), LRR (*least-recently-replaced*) ou *pseudo-random*.

As caches de instruções e de dados utilizadas neste projeto apresentam a seguinte configuração:

- Cache com 4 *ways*;
- Tamanho de cada *way*: 4 KiB;
- Tamanho de cada linha da cache: 16 bytes = 128 bits;
- Política de substituição LRU.

Na política de substituição LRU (*least-recently-used*), quando há um cache *miss*, a linha da cache que a mais tempo não é acessada é substituída pela nova linha lida da memória principal.

As caches de instruções e de dados também possuem algumas diferenças. Enquanto a cache de instruções possui um bit de validade para cada instrução de 32 bits, a cache de dados possui um único bit de validade para toda a linha da cache. Além disso, a política de escrita da cache de dados é *write-through* (o dado é escrito na cache e na memória principal).

2.1.2 Arquitetura

A Figura 2.1 apresenta a estrutura do *testbench* fornecido em conjunto com o código fonte do LEON3 e a sua divisão (em alto nível) em região segura e não segura. Neste nível de abstração, os blocos do *testbench* importantes para este trabalho são o processador LEON3 e o modelo da memória externa.

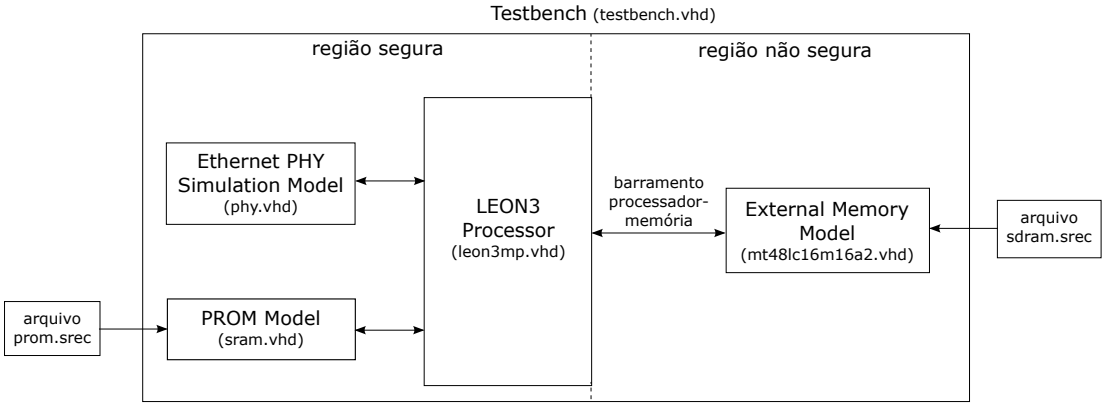

Figura 2.1: *Testbench* do LEON3 dividido em região segura e não segura.

O LEON3 é composto por AMBA AHB *masters* e *slaves*. Os AMBA AHB *masters* utilizados na configuração do LEON3 para este projeto são exibidos na Figura 2.2, e os AMBA AHB *slaves* na Figura 2.3. Na Figura 2.2, o barramento *ahbmi* são os sinais de entrada de um AMBA AHB *master*, e o barramento *ahbmo* são os sinais de saída de um AMBA AHB *master*. E, na Figura 2.3, o barramento *ahbsi* são os sinais de entrada de um AMBA AHB *slave*, e o barramento *ahbso* são os sinais de saída de um AMBA AHB *slave*.

Os principais blocos do *testbench* e do LEON3 para este projeto são:

- o controlador do barramento AMBA AHB (*AMBA AHB Controller*);
- o *core* do processador LEON3 (*High-performance SPARC V8 32-bit Processor*);
- o controlador de memória (*Combined PROM/IO/SRAM/SDRAM Memory Controller*);
- os PADs de entrada e de saída (*Input/Output PADs*);
- os PADs apenas de saída (*Output PADs*);

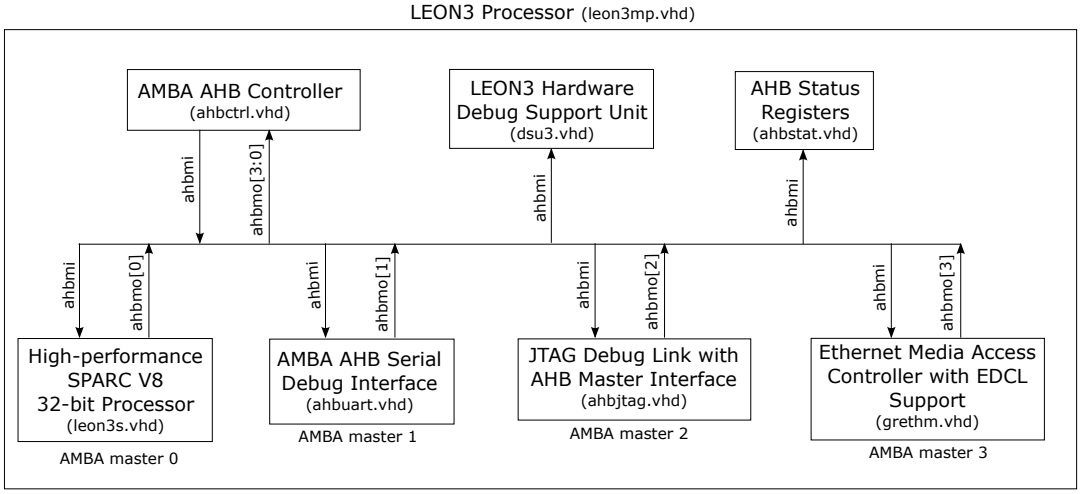

Figura 2.2: AMBA *masters* do LEON3.

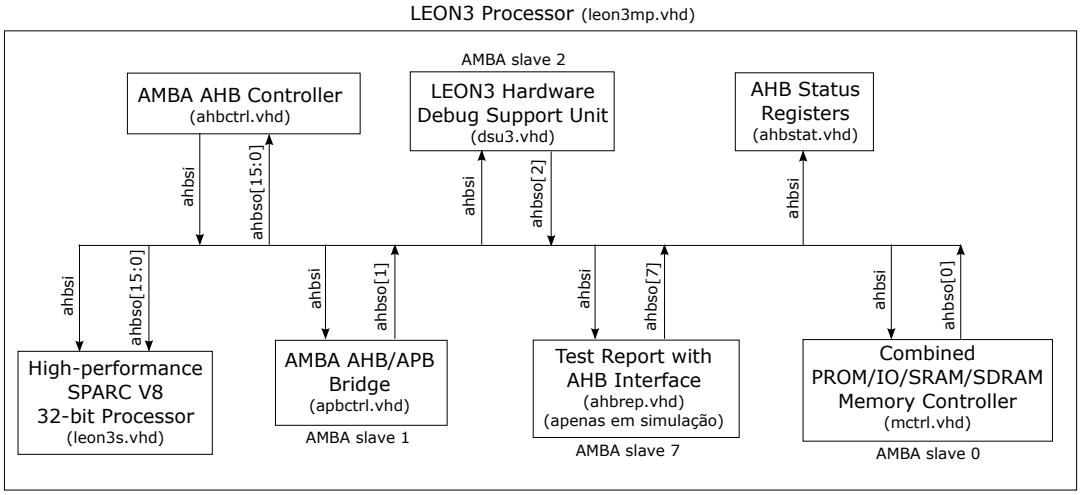

Figura 2.3: AMBA *slaves* do LEON3.

- o modelo da memória externa (*External Memory Model*).

A Figura 2.4 destaca estes blocos divididos em região segura e não segura. A região segura inclui o controlador do barramento AMBA AHB e o *core* do processador LEON3. Enquanto que a região não segura engloba o controlador de memória, os PADs e o modelo da memória externa.

A comunicação entre o controlador de memória e a memória externa (passando pelos PADs) depende da memória externa selecionada, isto é, da interface da memória externa e do seu modo de operação.

Por outro lado, a comunicação entre o *core* do LEON3 (AMBA *master* zero) e o controlador de memória (AMBA *slave* zero) é realizada através do protocolo AMBA AHB, ou seja, através de um protocolo de comunicação, portanto é independente da memória externa utilizada.

Assim, para que a camada de segurança (*security layer*) do LEON3-CONF fosse independente da memória externa, o limite entre a região segura e a não segura foi definido

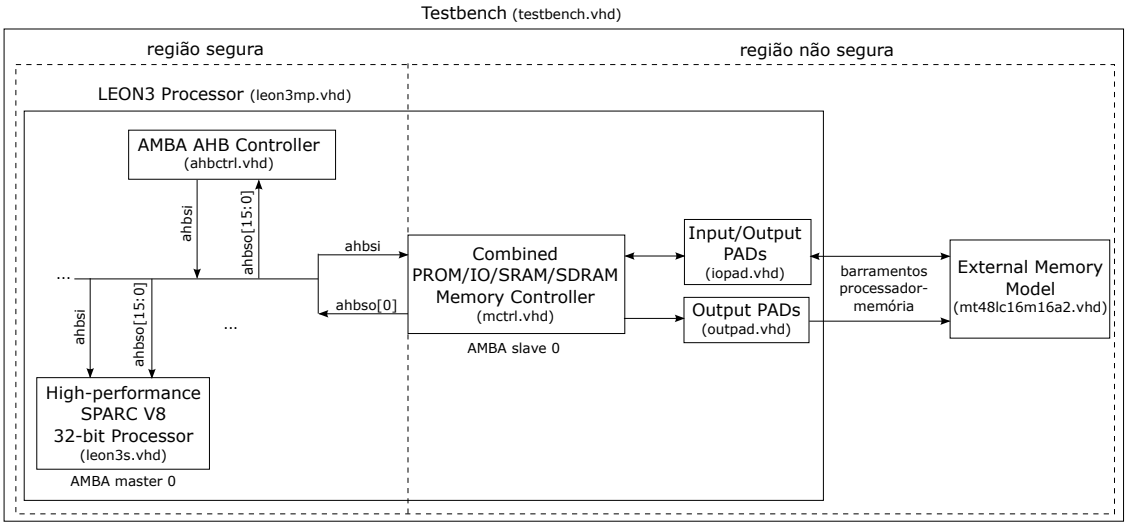

Figura 2.4: Blocos do *testbench* e do LEON3 divididos em região segura e não segura.

imediatamente antes do controlador de memória (Figura 2.4).

2.2 Algoritmo AES

O algoritmo de encriptação e decriptação selecionado para este projeto foi o algoritmo *Advanced Encryption Standard* (AES)[21, 37, 41].

A Figura 2.5 ilustra o diagrama de blocos da expansão da chave e da encriptação AES, e a Figura 2.6 o diagrama de blocos da expansão inversa da chave e da decriptação AES.

A Tabela 2.1 apresenta o número de *rounds*, o número de *S-boxes* e de *inverse S-boxes* utilizados em cada etapa do algortimo AES.

Tabela 2.1: Detalhes de cada etapa do algoritmo AES

Etapa do algoritmo AES	Número de *rounds*	Número de *S-boxes* ou *inverse S-boxes* com 32 bits de entrada
Expansão da chave	11 *rounds*	1 *S-box*
Encriptação AES	11 *rounds*	4 *S-boxes*
Expansão inversa da chave	11 *rounds*	1 *inverse S-box*
Decriptação AES	11 *rounds*	4 *inverse S-boxes*

Neste projeto, denomina-se *implementação direta do algoritmo AES*, a implementação que executa a expansão da chave seguida pela encriptação AES (Figura 2.7) e a expansão inversa da chave seguida pela decriptação AES (Figura 2.8).

A Tabela 2.2 destaca os detalhes da implementação direta do algoritmo AES considerando que cada *round* é executado em um período de clock, que a expansão da chave possui uma *S-box* dedicada (isto é, esta *S-box* não é utilizada pela encriptação AES), e

que a expansão inversa da chave possui uma *inverse S-box* dedicada (isto é, esta *inverse S-box* não é utilizada pela decriptação AES).

Tabela 2.2: Detalhes da implementação direta do algoritmo AES

Etapa do algoritmo AES	Número de períodos de clock	Número de *S-boxes* ou *inverse S-boxes* com 32 bits de entrada
Encriptação AES	22	5 *S-boxes*
Decriptação AES	22	5 *inverse S-boxes*

Com o intuito de reduzir o número de períodos de clock utilizados pelo algoritmo AES de encriptação e decriptação:

1. A encriptação AES é realizada em paralelo com a expansão da chave (Figura 2.9). Portanto, a encriptação AES utiliza 12 períodos de clock.

2. A decriptação AES é realizada em paralelo com a expansão inversa da chave (Figura 2.10). Entretanto, a decriptação AES pode utilizar:

   - 12 períodos de clock quando a chave do *round* 10 da expansão da chave está disponível, ou
   - 22 períodos de clock (Tabela 2.2) quando a chave do *round* 10 da expansão da chave não está disponível e, portanto, a expansão da chave precisa ser realizada antes da decriptação AES.

A Tabela 2.3 destaca o número de períodos de clock, de *S-boxes* e de *inverse S-boxes* utilizados pelo algoritmo AES de encriptação e decriptação implementados no LEON3-CONF.

Tabela 2.3: Detalhes do algoritmo AES implementado no LEON3-CONF

Etapa do algoritmo AES	Número de períodos de clock	Número de *S-boxes* ou *inverse S-boxes* com 32 bits de entrada
Encriptação AES	12	5 *S-boxes*
Decriptação AES quando a chave do *round* 10 da expansão da chave está disponível	12	5 *inverse S-boxes*
Decriptação AES quando a chave do *round* 10 da expansão da chave não está disponível	22	5 *inverse S-boxes*

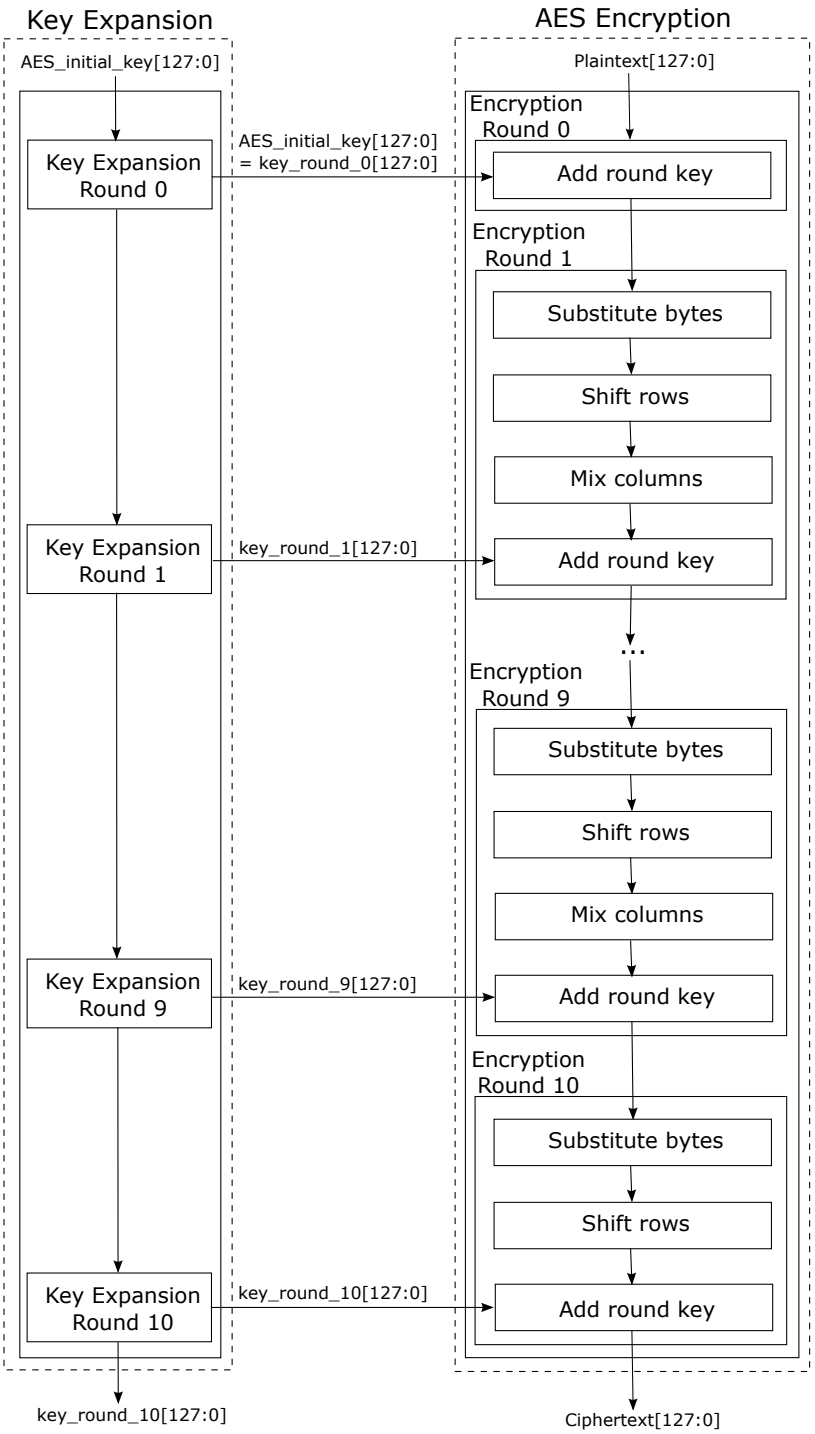

Figura 2.5: Diagrama de blocos da expansão da chave e da encriptação AES.

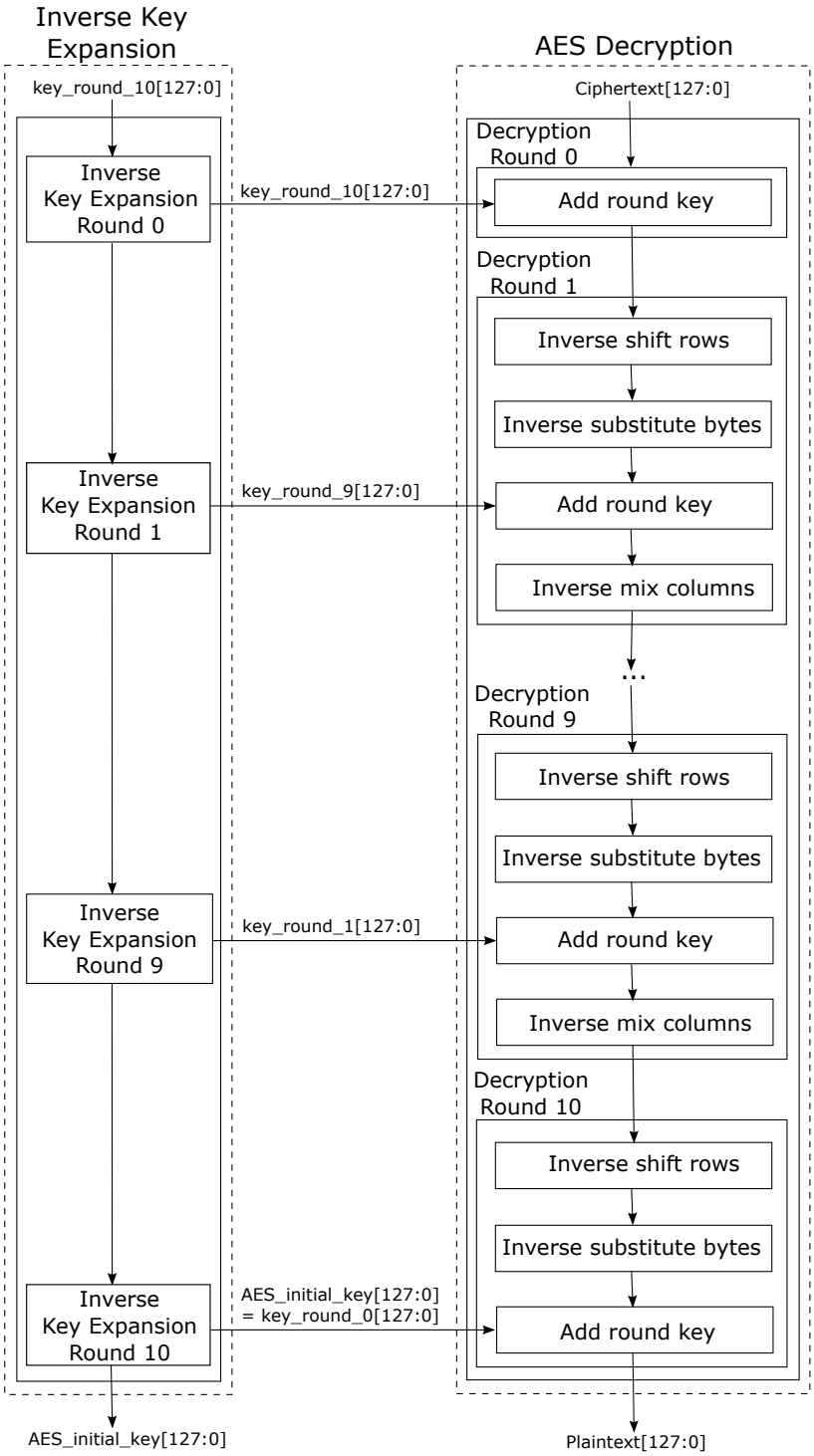

Figura 2.6: Diagrama de blocos da expansão inversa da chave e da decriptação AES.

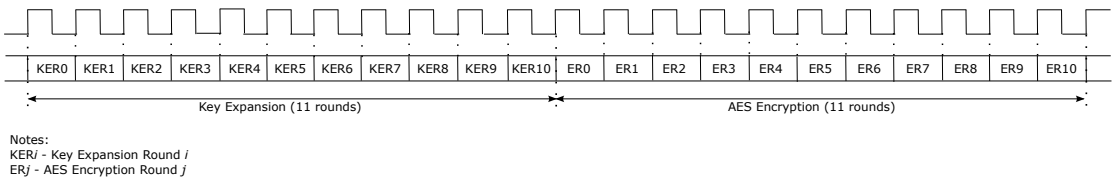

Figura 2.7: Diagrama de tempo da implementação direta da encriptação AES.

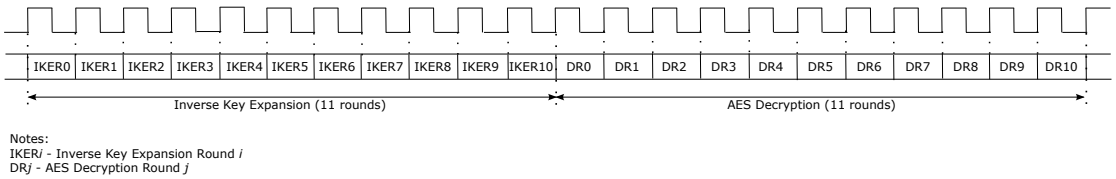

Figura 2.8: Diagrama de tempo da implementação direta da decriptação AES.

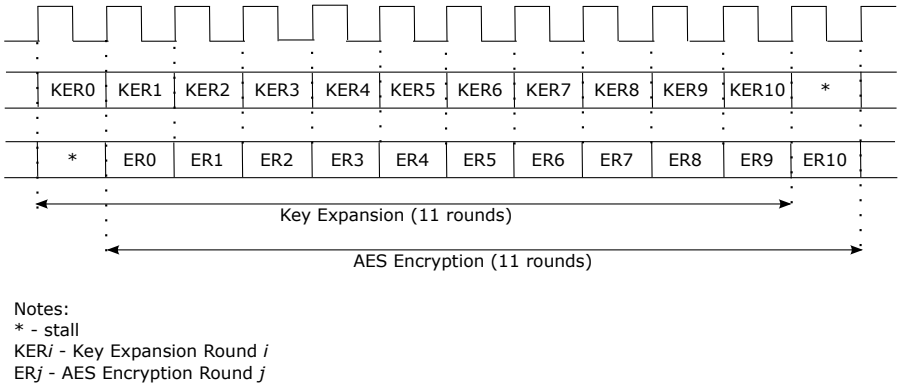

Figura 2.9: Diagrama de tempo da implementação da encriptação AES em paralelo com a expansão da chave.

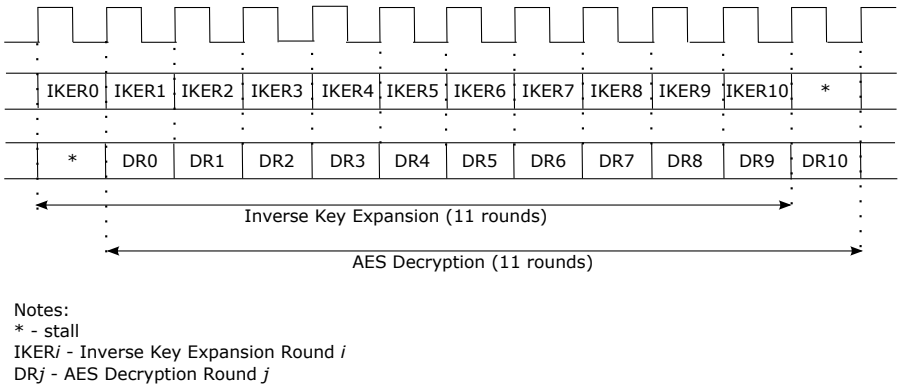

Figura 2.10: Diagrama de tempo da decriptação AES em paralelo com a expansão inversa da chave.

Capítulo 3

Arquitetura e Projeto do LEON3-CONF

Este capítulo apresenta o processador LEON3-CONF: processador LEON3 com confidencialidade de instruções e de dados. O capítulo está dividido em três seções. A Seção 3.1 detalha a camada de segurança, que é o bloco adicionado ao processador LEON3 para garantir a confidencialidade de instruções e/ou de dados e, então, obter o LEON3-CONF. Além disso, esta seção descreve dois blocos da camada de segurança: o módulo de registradores e a interface com o barramento AMBA AHB. A Seção 3.2, por sua vez, descreve o terceiro bloco da camada de segurança: a unidade de encriptação e decriptação. Por último, a Seção 3.3 documenta o desenvolvimento do RTL, a verificação e a síntese do LEON3-CONF.

3.1 Camada de Segurança

Esta seção apresenta a camada de segurança, o seu bloco que implementa os registradores (módulo de registradores) que são escritos e lidos pelos usuários, o seu bloco de interface com o barramento AMBA AHB, a padronização do cálculo do número de períodos de clock utilizados na execução de um programa no LEON3-CONF e o monitor usado para identificar determinadas transações AMBA AHB. A seção está estruturada em cinco divisões. A Seção 3.1.1 define o ponto de inclusão da camada de segurança no LEON3-CONF e apresenta a arquitetura da camada de segurança. A Seção 3.1.2 especifica em detalhes os nomes e a funcionalidade de todos registradores da camada de segurança implementados no módulo de registradores. A Seção 3.1.3, por sua vez, detalha a funcionalidade do bloco que faz a interface entre o *core* e a memória externa e vice-versa através do barramento AMBA AHB (bloco de interface com o barramento AMBA AHB) e documenta a máquina de estados implementada neste bloco. A Seção 3.1.4 informa que a camada de segurança também é capaz de calcular o número de períodos de clock utilizados na execução de um programa e padroniza este cálculo para a avaliação de desempenho do LEON3-CONF. Por último, a Seção 3.1.5 descreve o monitor de transações AMBA AHB projetado em VHDL para identificar as transações AMBA AHB que acessam instruções e/ou dados seguros. Estas transações identificadas são escritas em um arquivo texto e este monitor é

utilizado apenas quando o LEON3-CONF é simulado.

3.1.1 Visão Geral

A Figura 3.1 apresenta o ponto escolhido para incluir a camada de segurança (*security layer*) de acordo com o limite entre a região segura e a não segura definido na Figura 2.4.

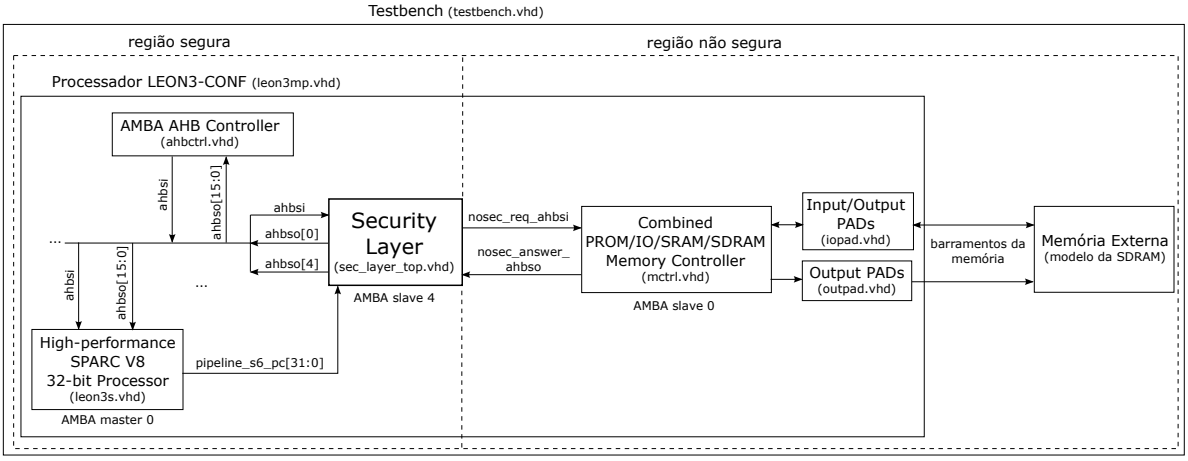

Figura 3.1: Ponto de inclusão da camada de segurança.

A Figura 3.2 ilustra o diagrama de blocos da camada de segurança. A camada de segurança é dividida em três blocos:

1. o módulo de registradores (*Registers Module*);
2. a interface com o barramento AMBA AHB (*Interface with the AMBA AHB Bus*);
3. a unidade de encriptação e de decriptação (*Ciphering and Deciphering Unit*).

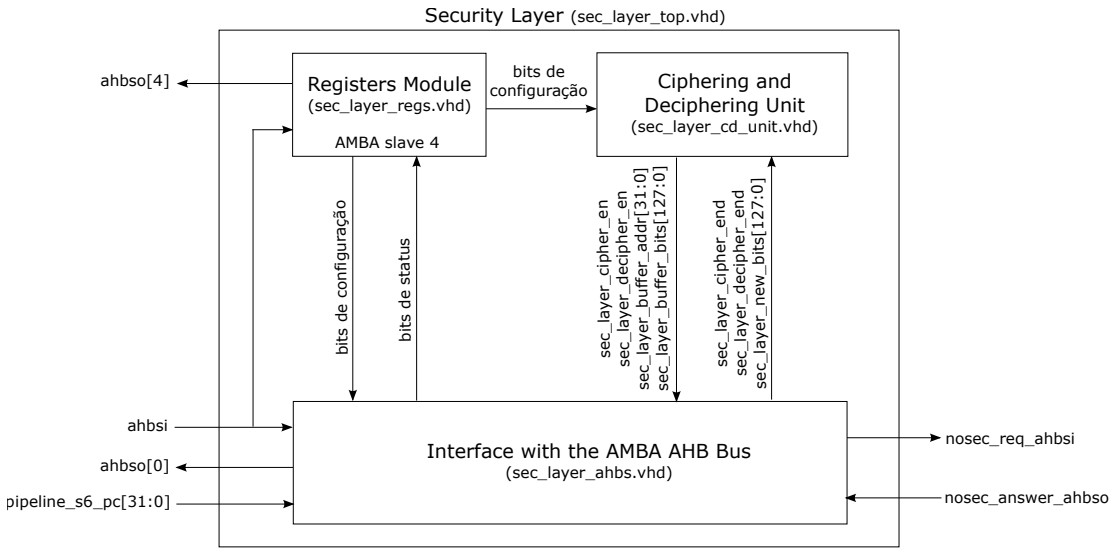

Figura 3.2: Diagrama de blocos da camada de segurança.

3.1.2 Módulo de Registradores

O módulo de registradores (*VHDL entity sec_layer_regs*) implementa os registradores da camada de segurança que são acessados pelos programas executados no LEON3-CONF. Estes registradores estão divididos de acordo com as suas funcionalidades em:

1. Registradores de configuração das regiões seguras;
2. Registradores de configuração da análise de performance;
3. Registradores com o resultado da análise de performance;
4. Registradores para o *debug* do LEON3-CONF programado em FPGA.

Registradores de Configuração das Regiões Seguras

Estes registradores configuram a utilização de regiões seguras da memória externa, a região da memória externa de dados seguros e a região da memória externa de instruções seguras. Os registradores deste grupo são listados na Tabela 3.1.

Estes registradores permitem que o programa a ser executado no processador LEON3-CONF também o configure. Assim, não é necessário sintetizar o LEON3-CONF a cada nova configuração do mesmo. Devido ao tempo da síntese do LEON3-CONF (Seção 3.3.3), a redução no número de sínteses agilizou a verificação, a validação e a análise de desempenho do LEON3-CONF em FPGA.

Tabela 3.1: Registradores de configuração das regiões seguras

Endereço	Nome do Registrador	*Read/Write*	Descrição
$B000_0000	SEC_CONFIG_REG	R/W	Configuração geral da camada de segurança
$B000_0010	SEC_DATA_CONFIG_REG	R/W	Configuração da região de dados seguros
$B000_0020	SEC_DATA_INIT_ADDR_REG	R/W	Endereço inicial da região de dados seguros
$B000_0024	SEC_DATA_FINAL_ADDR_REG	R/W	Endereço final da região de dados seguros
$B000_0030	SEC_DATA_KEY_WORD0_REG	R/W	Bits 127 a 96 da chave da região de dados seguros
$B000_0034	SEC_DATA_KEY_WORD1_REG	R/W	Bits 95 a 64 da chave da região de dados seguros
$B000_0038	SEC_DATA_KEY_WORD2_REG	R/W	Bits 63 a 32 da chave da região de dados seguros
$B000_003C	SEC_DATA_KEY_WORD3_REG	R/W	Bits 31 a 0 da chave da região de dados seguros

Continuação da Tabela 3.1

Endereço	Nome do Registrador	*Read/Write*	Descrição
$B000_0050	SEC_INST_ CONFIG_REG	R/W	Configuração da região de instruções seguras
$B000_0060	SEC_INST_INIT_ ADDR_REG	R/W	Endereço inicial da região de instruções seguras
$B000_0064	SEC_INST_FINAL_ ADDR_REG	R/W	Endereço final da região de instruções seguras
$B000_0070	SEC_INST_KEY_ WORD0_REG	R/W	Bits 127 a 96 da chave da região de instruções seguras
$B000_0074	SEC_INST_KEY_ WORD1_REG	R/W	Bits 95 a 64 da chave da região de instruções seguras
$B000_0078	SEC_INST_KEY_ WORD2_REG	R/W	Bits 63 a 32 da chave da região de instruções seguras
$B000_007C	SEC_INST_KEY_ WORD3_REG	R/W	Bits 31 a 0 da chave da região de instruções seguras

O registrador SEC_CONFIG_REG é detalhado na Tabela 3.2. O bit SEC_EN (*Security Layer Enable*) habilita a camada de segurança. Este bit deve ser setado apenas quando a configuração da camada de segurança estiver completa. Os demais bits deste registrador são reservados e lidos como zero.

Tabela 3.2: Registrador SEC_CONFIG_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SEC_EN

Os registradores com o prefixo "*SEC_DATA_*" configuram a região de dados seguros na memória externa.

O registrador SEC_DATA_CONFIG_REG é apresentado na Tabela 3.3. O bit SEC_DATA_EN (*Security Data Enable*) habilita uma região da memória externa como a região de dados seguros; o bit SEC_DATA_AES_EN (*Enable the algorithm AES for Security Data*) seleciona o algoritmo AES para encriptar e decriptar os dados seguros armazenados na memória externa; e os demais bits são reservados e lidos como zero.

Em trabalhos futuros, novos bits podem ser adicionados ao registrador SEC_DATA_ CONFIG_REG para selecionarem outros algoritmos de encriptação e decriptação para os dados seguros.

O registrador SEC_DATA_INIT_ADDR_REG define o endereço da memória externa onde inicia a região de dados seguros, e o registrador SEC_DATA_FINAL_ ADDR_REG o endereço da memória externa onde termina a região de dados seguros.

Tabela 3.3: Registrador SEC_DATA_CONFIG_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SEC_DATA_AES_EN
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SEC_DATA_EN

Os registradores SEC_DATA_KEY_WORD0_REG, SEC_DATA_KEY_WORD1_REG, SEC_DATA_KEY_WORD2_REG e SEC_DATA_KEY_WORD3_REG configuram a chave utilizada para encriptar e decriptar os dados da região segura.

Os registradores com o prefixo "*SEC_INST_*" configuram a região de instruções seguras na memória externa.

O registrador SEC_INST_CONFIG_REG é apresentado na Tabela 3.4. O bit SEC_INST_EN (*Security Instruction Enable*) habilita uma região da memória externa como a região das instruções seguras; o bit SEC_INST_AES_EN (*Enable the algorithm AES for Security Instructions*) seleciona o algoritmo AES para encriptar e decriptar as instruções seguras armazenadas na memória externa; e os demais bits são reservados e lidos como zero.

Em trabalhos futuros, novos bits podem ser adicionados ao registrador SEC_INST_CONFIG_REG para selecionarem outros algoritmos de encriptação e decriptação para as instruções seguras.

Tabela 3.4: Registrador SEC_INST_CONFIG_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SEC_INST_AES_EN
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SEC_INST_EN

O registrador SEC_INST_INIT_ADDR_REG define o endereço da memória externa onde inicia a região de instruções seguras, e o registrador SEC_INST_FINAL_ADDR_REG o endereço da memória externa onde termina a região de instruções seguras.

Os registradores SEC_INST_KEY_WORD0_REG, SEC_INST_KEY_WORD1_REG, SEC_INST_KEY_WORD2_REG e SEC_INST_KEY_WORD3_REG configuram a chave utilizada para encriptar e decriptar as instruções da região segura.

Em trabalhos futuros, recomenda-se que as chaves (registradores com os prefixos "*SEC_DATA_KEY_WORD*" e "*SEC_INST_KEY_WORD*") não possam ser acessadas (para leituras e nem para escritas) por programas ou só possam ser acessadas por programas com acesso privilegiado.

O Apêndice B ilustra a programação dos registradores deste grupo feita por um programa do *benchmark benchmark_c* (Seção 5.1.4).

Registradores de Configuração da Análise de Performance

Os registradores deste grupo configuram a análise de desempenho do LEON3-CONF a ser realizada durante a execução de um programa selecionado. Estes registradores são apresentados na Tabela 3.5 e devem ser escritos antes do bit SEC_EN ser setado.

Tabela 3.5: Registradores de configuração da análise de performance

Endereço	Nome do Registrador	*Read/Write*	Descrição
$B000_0090	SEC_PERF_CONFIG_REG	R/W	Configuração da análise de performance
$B000_00A0	SEC_PERF_INIT_ADDR_REG	R/W	Endereço inicial da análise de performance
$B000_00A4	SEC_PERF_FINAL_ADDR_REG	R/W	Endereço final da análise de performance

O registrador SEC_PERF_CONFIG_REG é mostrado na Tabela 3.6. O bit SEC_PERF_EN (*Security Performance Analysis Enable*) habilita a contagem do número de períodos de clock utilizados na execução de um programa e a indicação de quais estados e arcos da máquina de estados (implementada no arquivo *sec_layer_ahbs.vhd*) foram exercitados durante a execução deste programa. Os demais bits deste registrador são reservados e lidos como zero.

Tabela 3.6: Registrador SEC_PERF_CONFIG_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SEC_PERF_EN

O registrador SEC_PERF_INIT_ADDR_REG configura o endereço da memória com a instrução a partir da qual a contagem do número de períodos de clock será realizada, e o registrador SEC_PERF_FINAL_ADDR_REG estabelece o endereço da

memória com a instrução cuja execução encerra a contagem do número de períodos de clock (Seção 3.1.4).

O Apêndice B exemplifica a programação dos registradores deste grupo feita por um programa do *benchmark benchmark_ c* (Seção 5.1.4).

Registradores com o Resultado da Análise de Performance

Estes registradores armazenam o resultado da análise de performance do LEON3-CONF realizada durante a execução do programa selecionado. Os registradores deste grupo são listados na Tabela 3.7.

Tabela 3.7: Registradores com o resultado da análise de performance

Endereço	Nome do Registrador	*Read/Write*	Descrição
$B000_00B0	SEC_PERF_ STATUS_REG	R	Resultado da análise de performance
$B000_00B4	SEC_PERF_NUM_ CLOCKS_REG	R	Número de períodos de clock utilizados na execução de um programa
$B000_00B8	SEC_PERF_FSM_ CP_REG	R	Indicação dos estados e dos arcos da máquina de estados (arquivo *sec_ layer_ ahbs.vhd*) exercitados durante a execução de um programa

O registrador SEC_PERF_STATUS_REG é exibido na Tabela 3.8. O bit PERF_ END (*Security Performance Analysis End*) é setado quando a análise de performance do LEON3-CONF realizada durante a execução de um programa termina. Os demais bits do registrador SEC_PERF_STATUS_REG são reservados e lidos como zero.

Tabela 3.8: Registrador SEC_PERF_STATUS_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	PERF_END

O valor nos registradores SEC_PERF_NUM_CLOCKS_REG e SEC_PERF_ FSM_CP_REG são válidos quando o bit PERF_END é um.

O registrador SEC_PERF_NUM_CLOCKS_REG (*Security Performance - Number of Clocks*) contém o número de períodos de clock utilizados na execução do programa considerado.

O registrador SEC_PERF_FSM_CP_REG (*Security Performance - Finite State Machine Cover Points*) indica quais estados e arcos da máquina de estado (implementada no arquivo *sec_layer_ahbs.vhd*) foram exercitados durante a execução do programa considerado (Tabela 3.9).

Tabela 3.9: Registrador SEC_PERF_FSM_CP_REG

Bit	Estado	Condição ou arcos (da máquina de estados)
0	STATE_0	-
1	STATE_0	*if (new_sec_request_ff = '1') then*
2	STATE_0	*if (current_ahbsi_ff.hwrite = '1') then*
3	STATE_0	*if (current_ahbsi_ff.hwrite = '1') then - else*
4	STATE_0	*if (new_sec_request = '1') then*
5	STATE_0	*if (read_new_buffer_en = '0') then*
6	STATE_0	*if (sec_req_ahbsi.hwrite = '0') then*
7	STATE_0	*if (sec_req_ahbsi.hwrite = '0') then - else*
8	STATE_0	*elsif (buffer_was_modified = '0') then*
9	STATE_0	*elsif (buffer_was_modified = '0') then - else*
10	STATE_0	*if (buffer_was_modified = '1') then*
11	STATE_0	*if (buffer_was_modified = '1') then - else*
12	STATE_1	-
13	STATE_2	-
14	STATE_3	-
15	STATE_4	-
16	STATE_5	-
17	STATE_6	-
18	STATE_6	*if (current_ahbsi_ff.hwrite = '1') then*
19	STATE_6	*if (current_ahbsi_ff.hwrite = '1') then - else*
20	STATE_7	-
21	STATE_8	-
22	STATE_9	-
23	STATE_A	-
24	STATE_B	-
25	STATE_C	-
26	STATE_C	*if (empty_buffer_en_ff = '0') then*
27	STATE_C	*if (empty_buffer_en_ff = '0') then - else*
28	STATE_D	-
29	-	Bit não utilizado, por isso é lido como zero
30	-	Bit não utilizado, por isso é lido como zero
31	-	Bit não utilizado, por isso é lido como zero

Registradores para o *debug* do LEON3-CONF programado em FPGA

Os registradores deste grupo permitem a leitura do valor de alguns *flip-flops* da camada de segurança (implementada no arquivo *sec_layer_ahbs.vhd*) após a execução de um programa no LEON3-CONF. Estes registradores foram criados para auxiliar o *debug* do LEON3-CONF programado em FPGA.

Tabela 3.10: Registradores para o *debug* do LEON3-CONF programado em FPGA

Endereço	Nome do Registrador	*Read/Write*	Descrição
$B000_00C0	SEC_DEB_FSM_STATE_REG	R	Estado da máquina de estados
$B000_00C4	SEC_DEB_BUF_STATE_REG	R	Estado do *buffer* de 128 bits
$B000_00C8	SEC_DEB_BUF_ADDR_REG	R	Endereço da memória cujos bits estão no *buffer*
$B000_00D0	SEC_DEB_BUF_WORD0_REG	R	Bits 127 a 96 do *buffer*
$B000_00D4	SEC_DEB_BUF_WORD1_REG	R	Bits 95 a 64 do *buffer*
$B000_00D8	SEC_DEB_BUF_WORD2_REG	R	Bits 63 a 32 do *buffer*
$B000_00DC	SEC_DEB_BUF_WORD3_REG	R	Bits 31 a 0 do *buffer*

O registrador SEC_DEB_FSM_STATE_REG é detalhado na Tabela 3.11. O campo FSM_STATE[3:0] (*FSM state*) permite a leitura do estado final da máquina de estados (implementada no arquivo *sec_layer_ahbs.vhd*) depois da execução de um programa no LEON3-CONF. Os valores do campo FSM_STATE[3:0] estão na Tabela 3.14. Os demais bits do registrador SEC_DEB_FSM_STATE_REG são reservados e lidos como zero.

Tabela 3.11: Registrador SEC_DEB_FSM_STATE_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	FSM_STATE[3:0]			

O registrador SEC_DEB_BUF_STATE_REG é apresentado na Tabela 3.12. O campo BUFFER_STATUS[1:0] (*buffer status*) possibilita a leitura do estado final do *buffer* de 128 bits (utilizado no arquivo *sec_layer_ahbs.vhd* para armazenar o conteúdo de uma linha de 128 bits de uma das regiões seguras da memória externa) após a execução de um programa no LEON3-CONF. Os valores do campo BUFFER_STATUS[1:0]

estão na Tabela 3.13. Os demais bits do registrador SEC_DEB_BUF_STATE_REG são reservados e lidos como zero.

Tabela 3.12: Registrador SEC_DEB_BUF_STATE_REG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	BUFFER_ STATUS[1:0]	

Tabela 3.13: Valores do campo BUFFER_STATUS[1:0]

BUFFER_STATE[1:0]	Valor	Descrição
BUFFER_IS_EMPTY	0	O *buffer* está vazio
BUFFER_IS_VALID	1	O conteúdo do *buffer* é válido
BUFFER_WAS_MODIFIED	2	O conteúdo do *buffer* foi modificado

O registrador SEC_DEB_BUF_ADDR_REG permite a leitura do endereço da linha (de uma das regiões seguras da memória externa) armazenada no *buffer* (implementado no arquivo *sec_ layer_ ahbs.vhd*) depois da execução de um programa no LEON3-CONF.

Os registradores SEC_DEB_BUF_WORD0_REG, SEC_DEB_BUF_WORD1_ REG, SEC_DEB_BUF_WORD2_REG e SEC_DEB_BUF_WORD3_REG possibilitam a leitura do *buffer* de 128 bits (implementado no arquivo *sec_ layer_ ahbs.vhd*) que armazena o conteúdo de uma linha das regiões seguras da memória externa após a execução de um programa no LEON3-CONF.

3.1.3 Interface com o Barramento AMBA AHB

O bloco de interface com o barramento AMBA AHB (*VHDL entity sec_ layer_ ahbs*):

1. recebe os pedidos (de leitura ou de escrita) do *core* para a memória externa através do barramento AMBA AHB *ahbsi*;

2. verifica se a posição da memória externa a ser acessada pertence à região segura de dados ou à de instruções;

3. se a posição da memória externa a ser acessada não pertence a uma região segura, então:

   (a) o pedido é enviado à memória externa através do barramento AMBA AHB *nosec_ req_ ahbsi*, e

(b) o resultado fornecido pela memória externa (barramento AMBA AHB *nosec_answer_ahbso*) é enviado ao *core* através do barramento AMBA AHB *ahbso[0]*;

4. se a posição da memória externa a ser acessada pertence a uma região segura e o acesso é uma escrita, então:

   (a) o valor a ser escrito na memória externa é encriptado,

   (b) o pedido de escrita com o valor encriptado é enviado à memória externa através do barramento AMBA AHB *nosec_req_ahbsi*, e

   (c) o resultado da escrita fornecido pela memória externa (barramento AMBA AHB *nosec_answer_ahbso*) é enviado ao *core* pelo barramento AMBA AHB *ahbso[0]*;

5. se a posição da memória externa a ser acessada pertence a uma região segura e o acesso é uma leitura, então:

   (a) o pedido de leitura é enviado à memória externa através do barramento AMBA AHB *nosec_req_ahbsi*,

   (b) o valor lido que está encriptado é recebido da memória externa através do barramento AMBA AHB *nosec_answer_ahbso*,

   (c) o valor lido é decriptado, e

   (d) o valor decriptado é enviado ao *core* pelo barramento AMBA AHB *ahbso[0]*.

A Figura 3.3 apresenta a lógica entre os barramentos AMBA AHB *ahbsi* e *nosec_req_ahbsi*. Estes barramentos contém os pedidos (de leitura ou de escrita) do *core* para a memória externa.

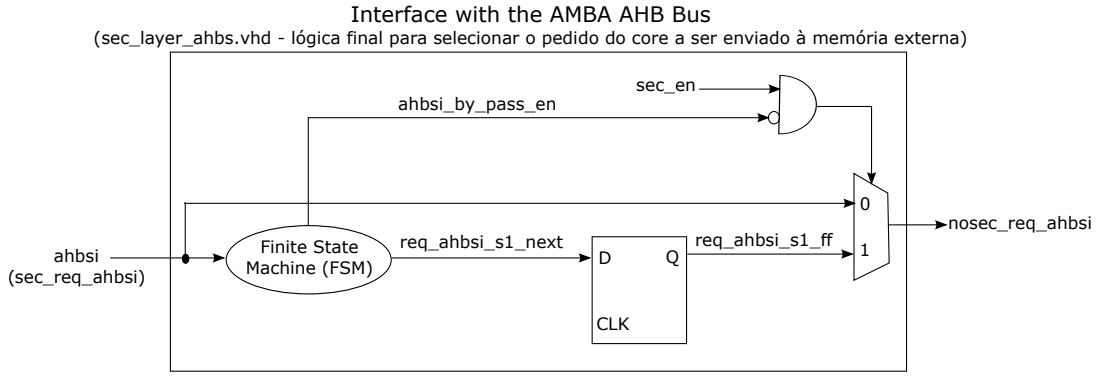

Figura 3.3: Lógica que seleciona o pedido do *core* a ser enviado à memória externa.

A Figura 3.4 esquematiza a lógica entre os barramentos AMBA AHB *nosec_answer_ahbso* e *ahbso[0]*. Estes barramentos contém as respostas da memória externa aos pedidos (de leitura ou de escrita) feitos pelo *core*.

As Figuras 3.3 e 3.4 destacam onde a máquina de estados implementada no arquivo *sec_layer_ahbs.vhd* foi incluída:

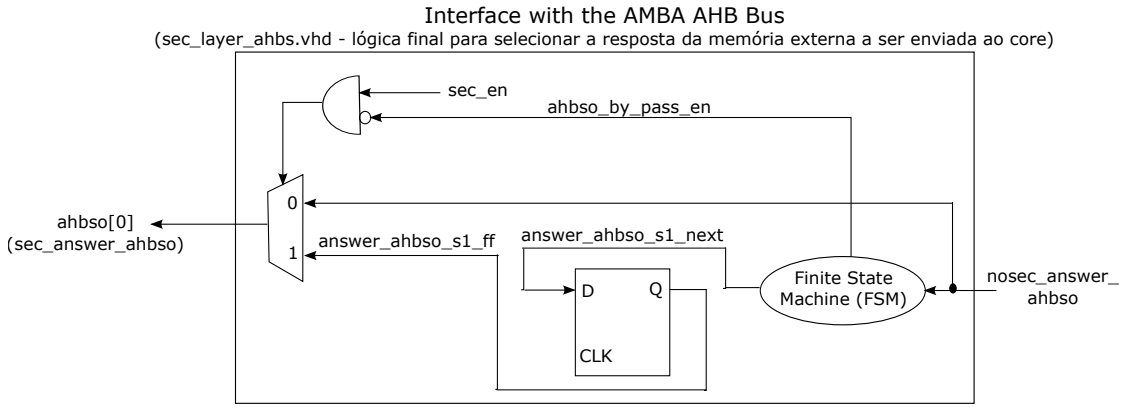

Figura 3.4: Lógica que seleciona a resposta da memória externa a ser enviada ao *core*.

- entre os barramentos AMBA AHB *ahbsi* e *nosec_req_ahbsi* (caminho 1) (Figura 3.3);
- entre os barramentos AMBA AHB *nosec_answer_ahbso* e *ahbso[0]* (caminho 2) (Figura 3.4).

Os caminhos 1 e 2 apresentaram violação de *timing* devido à inclusão da máquina de estados e, consequentemente, ao aumento significativo da lógica em cada um deles. Por isso, as saídas da máquina de estados foram amostradas por flip-flops denominados:

- *req_ahbsi_s1_ff* no caminho 1,
- *answer_ahbso_s1_ff* no caminho 2.

Nas Figuras 3.3 e 3.4, o sinal *sec_en* é o bit SEC_EN do registrador SEC_CONFIG_REG (Tabela 3.2).

A Figura 3.5 é o diagrama de estados da máquina de estados indicada nas Figuras 3.3 e 3.4 e implementada no arquivo *sec_layer_ahbs.vhd*.

A Tabela 3.14 contém a descrição dos 14 estados da máquina de estados da Figura 3.5. As transições entre os estados desta máquina são descritas na Tabela 3.15, e as ações decorrentes de determinadas transições na Tabela 3.16.

Tabela 3.14: Estados da máquina de estados implementada no arquivo *sec_layer_ahbs.vhd*

Estado	Valor	Descrição
STATE_0	0	Espera um novo pedido do barramento AMBA AHB *ahbsi* para a região das instruções ou dos dados seguros
STATE_1	1	Envia o pedido de leitura do endereço $xxxx_xxx0 para a memória externa
STATE_2	2	Envia o pedido de leitura do endereço $xxxx_xxx4 para a memória externa
STATE_3	3	Envia o pedido de leitura do endereço $xxxx_xxx8 para a

Continuação da Tabela 3.14

Estado	Valor	Descrição
		memória externa
STATE_4	4	Envia o pedido de leitura do endereço $xxxx_xxxC para a memória externa
STATE_5	5	Espera os 32 bits lidos do endereço $xxxx_xxxC estarem disponíveis no barramento *nosec_answer_ahbso*
STATE_6	6	Gera o sinal de *enable sec_layer_decipher_en* e espera até os 128 bits armazenados no *buffer* serem decriptados
STATE_7	7	Gera o sinal de *enable sec_layer_cipher_en* e espera até os 128 bits armazenados no *buffer* serem encriptados
STATE_8	8	Envia o pedido de escrita no endereço $xxxx_xxx0 da memória externa
STATE_9	9	Envia o pedido de escrita no endereço $xxxx_xxx4 da memória externa, e envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxx0 da memória externa
STATE_A	10	Envia o pedido de escrita no endereço $xxxx_xxx8 da memória externa, e envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxx4 da memória externa
STATE_B	11	Envia o pedido de escrita no endereço $xxxx_xxxC da memória externa, e envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxx8 da memória externa
STATE_C	12	Envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxxC da memória externa
STATE_D	13	Espera até o barramento AMBA AHB *ahbsi* não ter mais pedidos, isto é, estar no estado *idle*

Tabela 3.15: Descrição das transições entre os estados da máquina de estados implementada no arquivo *sec_layer_ahbs.vhd*

Transição	Descrição
T0.1	(Há um novo pedido no barramento AMBA AHB *ahbsi* para acessar uma das regiões seguras) e (o endereço da memória externa pedido não está no *buffer*) e (o conteúdo do *buffer* não foi modificado)
T0.2	(Há um novo pedido no barramento AMBA AHB *ahbsi* para acessar uma das regiões seguras) e (o endereço da memória externa pedido não está no *buffer*) e (o conteúdo do *buffer* foi modificado)
T0.3	(A última instrução do programa, cujo endereço na memória está armazenado no registrador SEC_PERF_FINAL_ADDR_REG[31:0], foi executada, isto é, está no estágio *Exception* do *pipeline*) e

Continuação da Tabela 3.15

Transição	Descrição
	(o conteúdo do *buffer* foi modificado)
T0.4	Senão
T1.1	(O pedido de leitura do endereço $xxxx_xxx0 foi recebido pela memória externa)
T1.2	Senão
T2.1	(Os 32 bits lidos do endereço $xxxx_xxx0 estão disponíveis no barramento *nosec_answer_ahbso*)
T2.2	Senão
T3.1	(Os 32 bits lidos do endereço $xxxx_xxx4 estão disponíveis no barramento *nosec_answer_ahbso*)
T3.2	Senão
T4.1	(Os 32 bits lidos do endereço $xxxx_xxx8 estão disponíveis no barramento *nosec_answer_ahbso*)
T4.2	Senão
T5.1	(Os 32 bits lidos do endereço $xxxx_xxxC estão disponíveis no barramento *nosec_answer_ahbso*)
T5.2	Senão
T6.1	(Os 128 bits armazenados no *buffer* foram decriptados)
T6.2	Senão
T7.1	(Os 128 bits armazenados no *buffer* foram encriptados)
T7.2	Senão
T8.1	(O pedido de escrita no endereço $xxxx_xxx0 foi recebido pela memória externa)
T8.2	Senão
T9.1	(32 bits encriptados foram escritos no endereço $xxxx_xxx0 da memória externa)
T9.2	Senão
TA.1	(32 bits encriptados foram escritos no endereço $xxxx_xxx4 da memória externa)
TA.2	Senão
TB.1	(32 bits encriptados foram escritos no endereço $xxxx_xxx8 da memória externa)
TB.2	Senão
TC.1	(32 bits encriptados foram escritos no endereço $xxxx_xxxC da memória externa) e (outros 128 bits da memória externa devem ser lidos)
TC.2	(32 bits encriptados foram escritos no endereço $xxxx_xxxC da memória externa) e (uma nova leitura da memória externa não precisa ser feita)
TC.3	Senão
TD.1	(O barramento AMBA AHB *ahbsi* está sem pedidos, isto é,

Continuação da Tabela 3.15

Transição	Descrição
	está no estado *idle*)
TD.2	Senão

Tabela 3.16: Descrição das ações decorrentes das transições de estados da máquina de estados implementada no arquivo *sec_layer_ahbs.vhd*

Ação	Descrição
A0.1	(Envia o pedido de leitura do endereço $xxxx_xxx0 para a memória externa)
A1.1	(Envia o pedido de leitura do endereço $xxxx_xxx4 para a memória externa)
A2.1	(Envia o pedido de leitura do endereço $xxxx_xxx8 para a memória externa) e (armazena os 32 bits lidos e encriptados no *buffer* - do bit 127 ao 96)
A3.1	(Envia o pedido de leitura do endereço $xxxx_xxxC para a memória externa) e (armazena os 32 bits lidos e encriptados no *buffer* - do bit 95 ao 64)
A4.1	(Armazena os 32 bits lidos e encriptados no *buffer* - do bit 63 ao 32)
A5.1	(Armazena os 32 bits lidos e encriptados no *buffer* - do bit 31 ao 0)
A6.1	(Armazena os 128 bits decriptados no *buffer*) e (envia a resposta do pedido de leitura ou de escrita ao *core*)
A7.1	(Armazena os 128 bits encriptados no *buffer*) e (envia o pedido de escrita no endereço $xxxx_xxx0 da memória externa)
A8.1	(Envia o pedido de escrita no endereço $xxxx_xxx4 da memória externa) e (envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxx0 da memória externa)
A9.1	(Envia o pedido de escrita no endereço $xxxx_xxx8 da memória externa) e (envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxx4 da memória externa)
AA.1	(Envia o pedido de escrita no endereço $xxxx_xxxC da memória externa) e (envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxx8 da memória externa)
AB.1	(Envia os 32 bits encriptados a serem escritos no endereço $xxxx_xxxC da memória externa)
AC.1	(Envia o pedido de leitura do endereço $xxxx_xxx0 para a memória externa)

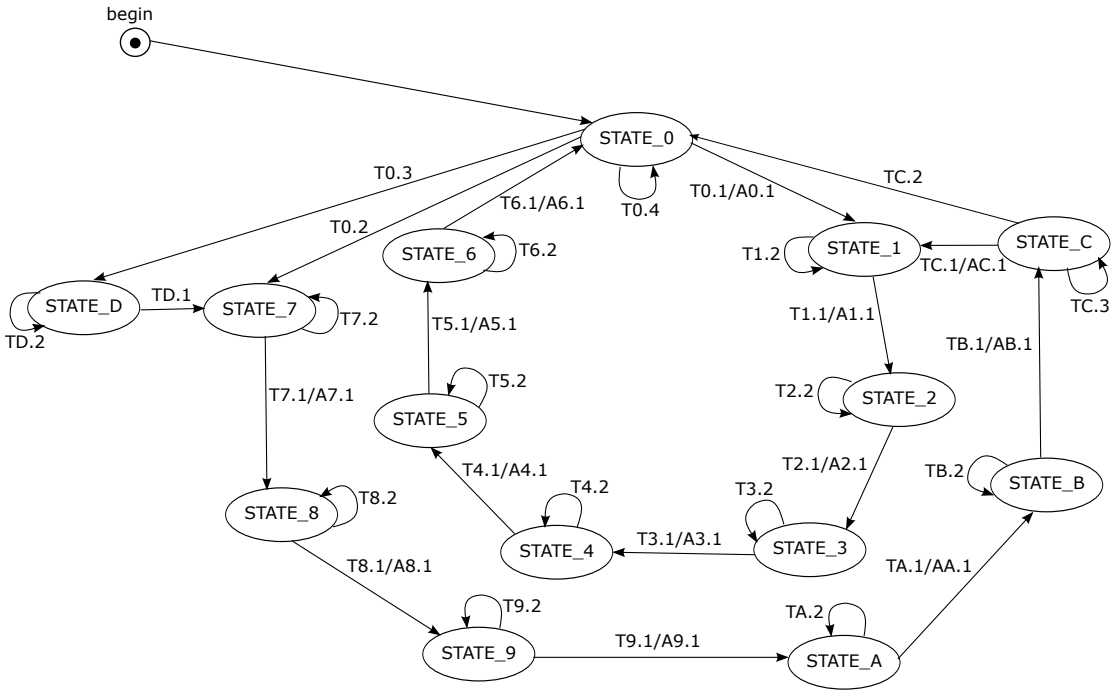

Figura 3.5: Diagrama de estados da máquina de estados em *sec_ layer_ ahbs.vhd*.

É importante destacar que os estados STATE_6 e STATE_7 desta máquina de estados são independentes do tempo que os algoritmos de encriptação e decriptação (implementados na unidade de encriptação e decriptação - Seção 3.2) precisam para encriptar e decriptar 128 bits, respectivamente.

Isto é possível porque o STATE_6 espera até a unidade de encriptação e decriptação indicar que os 128 bits foram decriptados (através do sinal *sec_ layer_ decipher_ end*), e o STATE_7 espera até a mesma unidade indicar que os 128 bits foram encriptados (através do sinal *sec_ layer_ cipher_ end*).

3.1.4 Medida do Tempo de Execução de um Programa

A máquina de estados (Figura 3.5) implementada na interface com o barramento AMBA AHB (Seção 3.1.3) também conta o número de períodos de clock utilizados na execução de um programa.

Esta contagem inicia quando o bit SEC_PERF_EN (do registrador SEC_PERF_CONFIG_REG) é um e o endereço da instrução a ser lida da memória, isto é, o endereço no barramento *ahbsi* (Figura 3.1) é igual ao endereço configurado no registrador SEC_PERF_INIT_ADDR_REG.

Esta contagem termina quando o endereço da instrução no estágio *Exception* (estágio 6) do *pipeline* do processador (sinal *pipeline_ s6_ pc[31:0]* na Figura 3.1) é igual ao endereço configurado no registrador SEC_PERF_FINAL_ADDR_REG.

Para padronizar a contagem dos períodos de clock durante a execução de um programa, as seguintes considerações foram adotadas:

1. Em relação ao endereço de memória a ser configurado no registrador SEC_PERF_INIT_ADDR_REG:

- As instruções que configuram os registradores da camada de segurança não são incluídas na análise de desempenho.
- Entre a última instrução que configura os registradores da camada de segurança e a primeira instrução do programa em si, deve-se ter no mínimo 4 instruções do tipo *nop*.
- O endereço a ser escrito no registrador SEC_PERF_INIT_ADDR_REG é o endereço de memória da primeira instrução do programa após as 4 instruções *nop*.

2. Em relação ao endereço de memória a ser configurado no registrador SEC_PERF_FINAL_ADDR_REG:

   - Após a última instrução do programa em si, deve-se ter no mínimo 8 instruções do tipo *nop*.
   - O endereço a ser escrito no registrador SEC_PERF_FINAL_ADDR_REG é o endereço de memória da primeira instrução *nop* após a última instrução do programa em si.

Ao final da execução de um programa com a análise de desempenho habilitada (isto é, o bit SEC_PERF_EN é um), o bit PERF_END está setado e o registrador SEC_PERF_NUM_CLOCKS_REG (*Security Performance - Number of Clocks*) contém o número de períodos de clock utilizados na execução deste programa.

3.1.5 Monitor de Transações AMBA AHB

O monitor de transações AMBA AHB (*VHDL entity ahbs_monitor_top*) foi desenvolvido para auxiliar na verificação da camada de segurança (Seção 3.1).

A Figura 3.6 destaca o ponto onde o monitor de transações AMBA AHB (*AMBA AHB monitor*) foi incluído na arquitetura do LEON3-CONF (Figura 3.1).

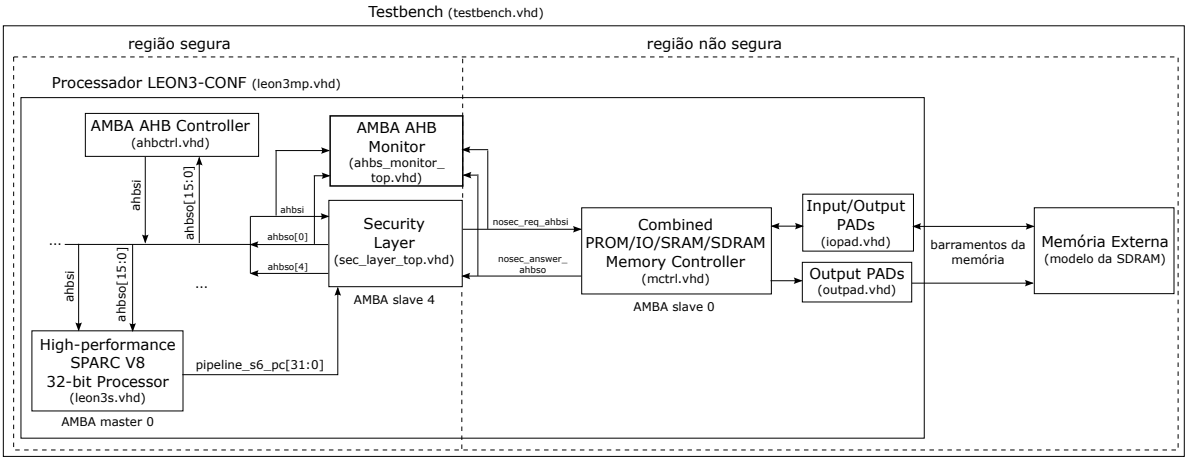

Figura 3.6: Ponto de inclusão do monitor de transações AMBA AHB.

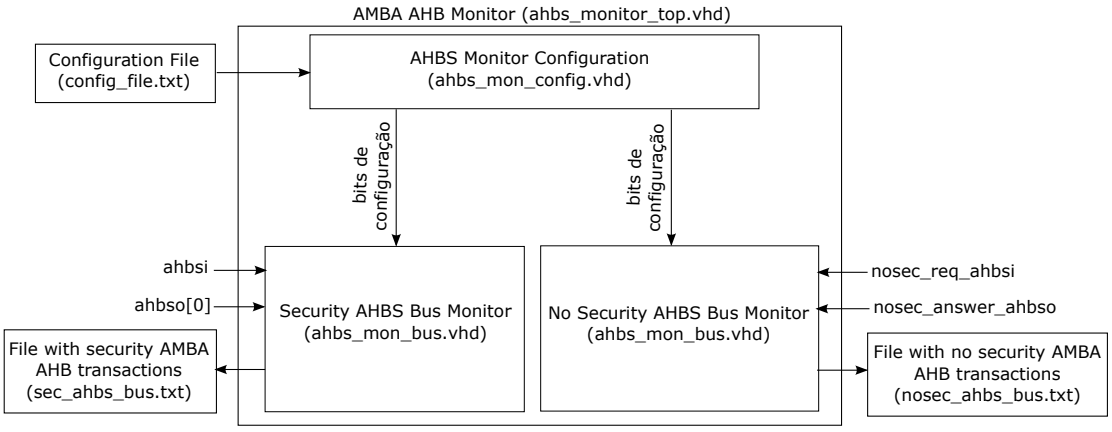

Figura 3.7: Arquitetura do monitor de transações AMBA AHB.

A Figura 3.7 detalha a arquitetura do monitor de transações AMBA AHB. O bloco *AHBS Monitor Configuration* (*VHDL entity ahbs_mon_config*) lê o arquivo de configuração (*config_file.txt*), que é o mesmo arquivo de configuração lido pelo programa *srec_cipher.c* (Seção 5.2.4), e envia as informações às duas instâncias do bloco *AHBS Bus Monitor* (*VHDL entity ahbs_mon_bus*):

- *Security AHBS Bus Monitor* (nome da instância: *ahbs_mon_sec_bus_inst*) e
- *No Security AHBS Bus Monitor* (nome da instância: *ahbs_mon_nosec_bus_inst*).

O bloco *AHBS Bus Monitor* monitora o barramento AMBA AHB com o objetivo de identificar as transações AMBA AHB para a região de instruções seguras e/ou para a de dados seguros de acordo com a configuração selecionada pelo arquivo de configuração. Quando tais transações são encontradas, o endereço da posição da memória acessada, o tipo da transação (leitura ou escrita) e o dado (lido ou escrito) são escritos em um arquivo texto.

A seguir há um exemplo do arquivo texto gerado pelo módulo *AHBS Bus Monitor* com duas transações AMBA AHB.

```
Address: 4000021C - Read value:  92F696EE
Address: 40000210 - Write value: AD663330
```

A instância *Security AHBS Bus Monitor* monitora as transações dos barramentos AMBA AHB seguros (barramentos *ahbsi* e *ahbso[0]* nas Figuras 3.6 e 3.7). Por isso, as instruções e os dados seguros estão na forma clara nestes barramentos. O nome do arquivo texto gerado por esta instância do bloco *AHBS Bus Monitor* é *sec_ahbs_bus.txt*.

A instância *No Security AHBS Bus Monitor* monitora as transações dos barramentos AMBA AHB não seguros (barramentos *nosec_req_ahbsi* e *nosec_answer_ahbso* nas Figuras 3.6 e 3.7). Por esta razão, as instruções e os dados seguros estão encriptados nestes barramentos. O nome do arquivo texto gerado por esta instância do bloco *AHBS Bus Monitor* é *nosec_ahbs_bus.txt*.

A Tabela 3.17 contém os arquivos utilizados e gerados pelo monitor de transações AMBA AHB.

Tabela 3.17: Arquivos utilizados e gerados pelo monitor de transações AMBA AHB

Arquivo(s)	Diretório
Código VHDL do monitor	*projectrepository/testbench/monitors/ahbs_monitor/vhdl_code*
config_file.txt	*projectrepository/testbench/monitors/ahbs_monitor/config_files*
sec_ahbs_bus.txt	*projectrepository/testbench/monitors/ahbs_monitor/result_files*
nosec_ahbs_bus.txt	*projectrepository/testbench/monitors/ahbs_monitor/result_files*

3.2 Unidade de Encriptação e Decriptação

Esta seção detalha a unidade de encriptação e decriptação instanciada na camada de segurança (Seção 3.1) do LEON3-CONF. A seção está estruturada em três divisões. A Seção 3.2.1 apresenta a arquitetura da unidade de encriptação e decriptação. A Seção 3.2.2 descreve o bloco *AES Top* que implementa o algoritmo AES. Por último, a Seção 3.2.3 documenta a verificação do algoritmo AES (bloco *AES Top*).

3.2.1 Visão Geral

A Figura 3.8 ilustra o diagrama de blocos da unidade de encriptação e decriptação (*VHDL entity sec_layer_cd_unit*).

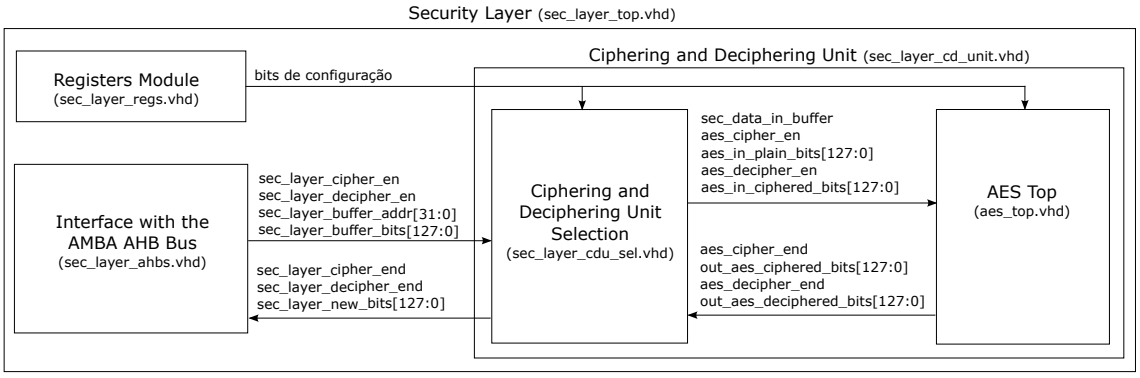

Figura 3.8: Diagrama de blocos da unidade de encriptação e decriptação.

A unidade de encriptação e decriptação é dividida em dois blocos:

1. o módulo de seleção da unidade de encriptação e decriptação (*Ciphering and Deciphering Unit Selection*);

2. o módulo que implementa o algoritmo AES de encriptação e decriptação (*AES Top*).

O módulo de seleção da unidade de encriptação e decriptação (*VHDL entity sec_layer_cdu_sel*) faz a conexão entre a interface com o barramento AMBA AHB (Se-

ção 3.1.3) e o módulo que implementa a encriptação e decriptação AES (*VHDL entity aes_top*).

Mais detalhadamente, o módulo de seleção da unidade de encriptação e decriptação gera os seguintes sinais:

- *sec_data_in_buffer* que indica quando o *buffer* contém instruções seguras ou dados seguros;
- *aes_cipher_en* que seleciona o algoritmo AES de encriptação;
- *aes_in_plain_bits[127:0]* que contém os 128 bits a serem encriptados pelo algoritmo AES;
- *aes_decipher_en* que seleciona o algoritmo AES de decriptação;
- *aes_in_ciphered_bits[127:0]* que contém os 128 bits a serem decriptados pelo algoritmo AES;
- *sec_layer_cipher_end* que indica o fim da encriptação AES;
- *sec_layer_decipher_end* que indica o fim da decriptação AES;
- *sec_layer_new_bits[127:0]* que contém o resultado da encriptação AES ou da decriptação AES.

Em trabalhos futuros, outros algoritmos de encriptação e decriptação podem ser implementados em blocos separados e serem instanciados na unidade de encriptação e decriptação (*sec_layer_cd_unit*).

3.2.2 Implementação do Algoritmo AES

A Figura 3.9 contém o diagrama de blocos do bloco que implementa o algoritmo AES (*VHDL entity aes_top*) de acordo com a estratégia apresentada no Seção 2.2. O bloco que implementa o algoritmo AES é dividido em cinco blocos:

1. o bloco que implementa a expansão da chave (*AES Key Expansion*);
2. o bloco que implementa a encriptação AES (*AES Encryption*);
3. o bloco que implementa a expansão inversa da chave (*AES Inverse Key Expansion*);
4. o bloco que implementa a decriptação AES (*AES Decryption*);
5. o bloco que controla a execução dos quatro blocos anteriores (*AES Control*).

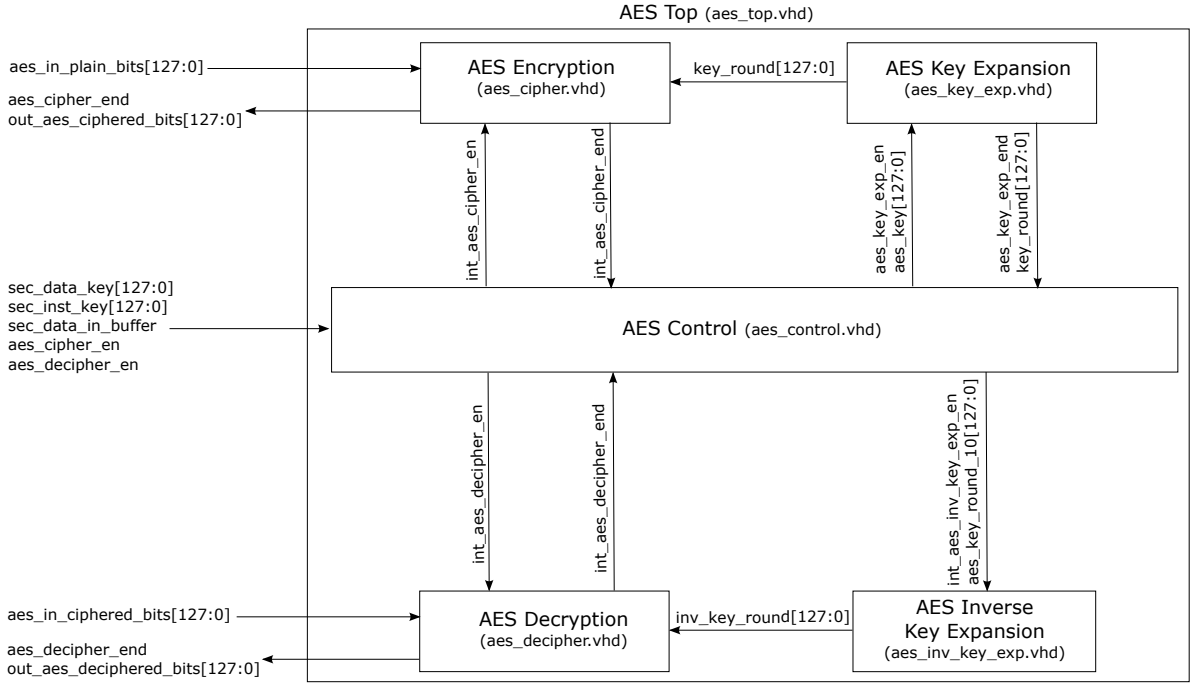

Figura 3.9: Diagrama de blocos do bloco que implementa o algoritmo AES.

O bloco *AES Key Expansion* (*VHDL entity aes_key_exp*) implementa uma máquina de estados com doze estados. Cada um dos primeiros onze estados (do estado STATE_0 ao estado STATE_A) é um *round* da expansão da chave (*Key Expansion Round* na Figura 2.5). Ao final de cada um destes estados, a chave gerada é armazenada em 128 flip-flops e é utilizada como entrada no próximo estado. O último estado (STATE_B) indica que a expansão da chave foi concluída (sinal *aes_key_exp_end*).

O bloco *AES Encryption* (*VHDL entity aes_cipher*) implementa uma máquina de estados com treze estados. O primeiro estado (STATE_0) é o estado anterior ao *round 0* da encriptação AES (*Encryption Round 0* na Figura 2.5). O objetivo deste estado é apenas armazenar os bits que serão encriptados (*plaintext[127:0]*) em flip-flops. Cada um dos próximos onze estados (do estado STATE_1 ao estado STATE_B) é um *round* da encriptação AES (*Encryption Round*). Ao final de cada um destes estados, os 128 bits gerados são armazenados em flip-flops e são utilizados como entrada no próximo estado. O último estado (STATE_C) indica que a encriptação AES foi concluída (sinal *aes_cipher_end*).

O bloco *AES Inverse Key Expansion* (*VHDL entity aes_inv_key_exp*) implementa uma máquina de estados com doze estados. Cada um dos primeiros onze estados (do estado STATE_0 ao estado STATE_A) é um *round* da expansão inversa da chave (*Inverse Key Expansion Round* na Figura 2.6). Ao final de cada um destes estados, a chave gerada é armazenada em 128 flip-flops e é utilizada como entrada no próximo estado. O último estado (STATE_B) indica que a expansão inversa da chave foi concluída (sinal *aes_inv_key_exp_end*).

O bloco *AES Decryption* (*VHDL entity aes_decipher*) implementa uma máquina de estados com treze estados. O primeiro estado (STATE_0) é o estado anterior ao *round 0* da decriptação AES (*Decryption Round 0* na Figura 2.6). O objetivo deste estado é

apenas armazenar os bits que serão decriptados (*ciphertext[127:0]*) em flip-flops. Cada um dos próximos onze estados (do estado STATE_1 ao estado STATE_B) é um *round* da decriptação AES (*Decryption Round*). Ao final de cada um destes estados, os 128 bits gerados são armazenados em flip-flops e são utilizados como entrada no próximo estado. O último estado (STATE_C) indica que a decriptação AES foi concluída (sinal *aes_decipher_end*).

A estrutura dos blocos *AES Key Expansion* e *AES Inverse Key Expansion* são similares devido à simetria dos seus algoritmos. A mesma análise é válida para os blocos *AES Encryption* e *AES Decryption*.

O bloco *AES Control* (*VHDL entity aes_control*) implementa a máquina de estados (Figura 3.10) que controla a execução das máquinas de estados dos outros quatro blocos (Figura 3.9).

A função de cada um dos estados da máquina de estados do bloco *AES Control* é:

- O STATE_0 é o estado de espera, isto é, a execução da máquina de estados permanece neste estado até receber um comando de encriptação AES (*aes_cipher_en*) ou um de decriptação AES (*aes_decipher_en*).
- O STATE_1 controla a execução em paralelo da expansão da chave com a encriptação AES (Figura 2.9). Neste estado, a chave resultante do *round* 10 da expansão da chave das instruções seguras e dos dados seguros são armazenadas em flip-flops. Estas chaves são utilizadas na expansão inversa da respectiva chave.
- O STATE_2 controla a execução da expansão da chave caso a decriptação AES tenha sido selecionada e a correspondente chave do *round* 10 não esteja disponível.
- O STATE_3 controla a execução da expansão inversa da chave em paralelo com a decriptação AES (Figura 2.10).

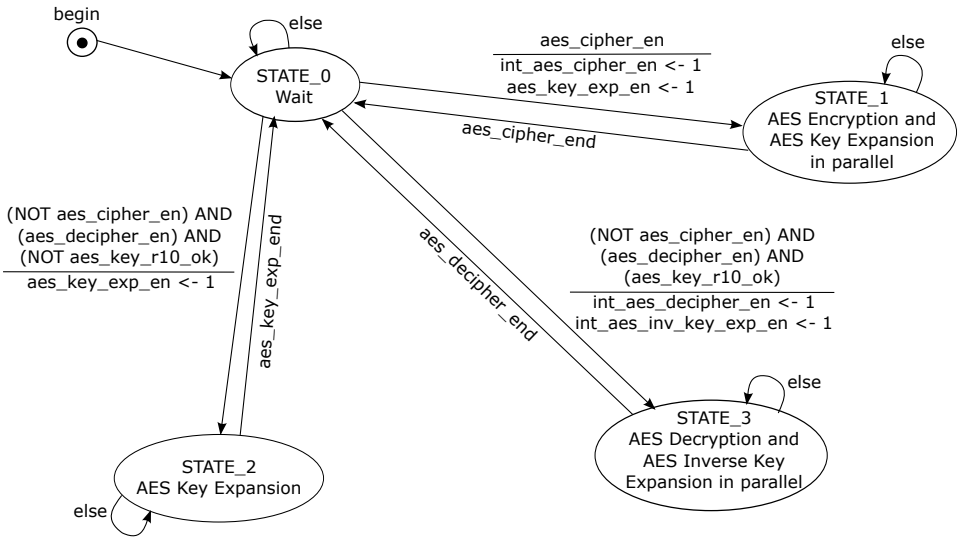

Figura 3.10: Diagrama de estados da máquina de estados no bloco *AES Control*.

3.2.3 Verificação do Algoritmo AES

Esta seção detalha a verificação do bloco que implementa o algoritmo AES (Seção 3.2.2). Esta verificação foi dividida em duas etapas. Na primeira etapa, o algoritmo AES propriamento dito é verificado. Na segunda etapa, a máquina de estados implementada no bloco *AES Control* (Figura 3.10) para controlar as fases do algoritmo AES (expansão da chave, encriptação AES, expansão inversa da chave e decriptação AES) é verificada. Estas duas etapas da verificação são descritas na sequência.

Primeira Etapa da Verificação do Algoritmo AES

Esta etapa verifica a implementação do algoritmo AES de encriptação e decriptação através de simulações. A Figura 3.11 apresenta o diagrama de blocos do *testbench* *aes_dev_nist_tests_tb*.

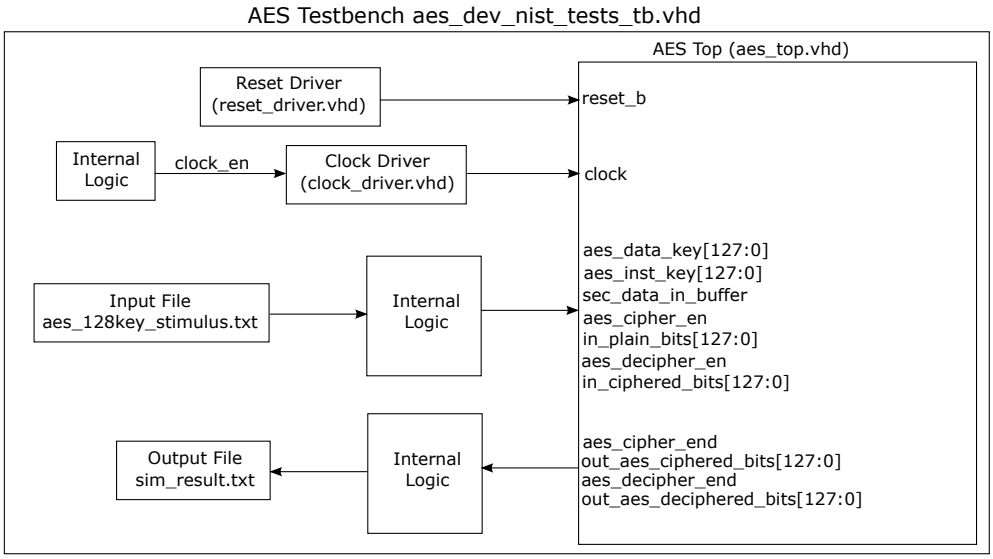

Figura 3.11: Diagrama de blocos do *testbench aes_dev_nist_tests_tb*.

O arquivo *projectrepository/testbench/stimulus/aes_128key_stimulus.txt* contém os 128 testes fornecidos pelo arquivo *ecb_tbl.txt* (*rijndael-vals.zip*[37]). Cada um destes testes é composto por uma chave de 128 bits, 128 bits claros e 128 bits encriptados. Cada teste é simulado duas vezes: na primeira vez, os bits claros são encriptados e, na segunda, os bits encriptados são decriptados. Os resultados das duas simulações são comparados com os valores esperados, e os resultados das comparações são escritos no arquivo *projectrepository/testbench/result/aes_dev_nist_tests_tb/sim_result.txt*.

Os scripts *bash* desenvolvidos para executar o GHDL estão descritos na Tabela 3.18 e estão no diretório *projectrepository/tool_data/ghdl/aes/aes_dev_nist_tests_tb*.

Nesta etapa, os 128 testes (cada um com uma encriptação AES e uma decriptação AES) passaram.

Tabela 3.18: Scripts desenvolvidos para simular o *testbench aes_dev_nist_tests_tb*

Nome do script	Descrição do script
script_00_create_dirs.sh	Este script remove e cria os diretórios utilizados na execução do GHDL.
script_01_ghdl_import.sh	Este script lista e importa todos os arquivos VHDL que compõem o *testbench* e o *aes_top.vhd*.
script_02_ghdl_make.sh	Este script *analyze* e *elaborate* o *testbench* e o *aes_top.vhd*.
script_03_ghdl_generate_wave.sh	Este script simula o *testbench* e o *aes_top.vhd* e grava a forma de onda em *projectrepository/work/waveform/aes/aes_dev_nist_tests_tb/aes_dev_nist_tests_tb.vcd*.

Segunda Etapa da Verificação do Algoritmo AES

Esta etapa verifica a máquina de estados implementada no bloco *AES Control* (Figura 3.10). A Figura 3.12 exibe o diagrama de blocos do *testbench aes_ctrl_tb*.

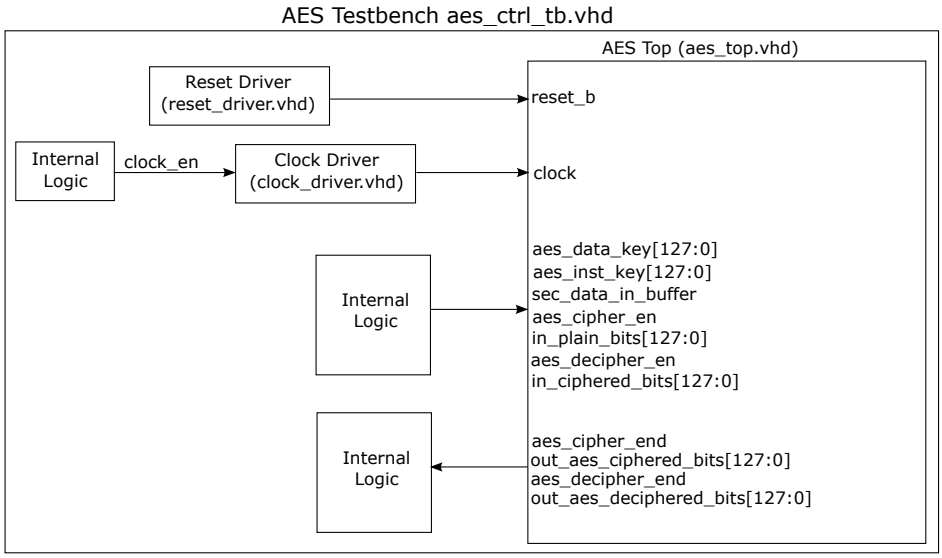

Figura 3.12: Diagrama de blocos do *testbench aes_ctrl_tb*.

O objetivo do *testbench aes_ctrl_tb* é estimular o bloco *aes_top* de modo que todos os estados e arcos da máquina de estados no bloco *AES Control* sejam exercitados. Para tanto, o *testbench* executa cinco testes (dois de encriptação AES e três de decriptação AES) e alterna entre instruções seguras e dados seguros.

Além de estimular o bloco *aes_top*, o *testbench aes_ctrl_tb* verifica se os resultados do *aes_top* são iguais aos esperados.

Os scripts *bash* desenvolvidos para executar o GHDL estão descritos na Tabela 3.19 e estão no diretório *projectrepository/tool_data/ghdl/aes/aes_ctrl_tb*.

Tabela 3.19: Scripts desenvolvidos para simular o *testbench aes_ctrl_tb*

Nome do script	Descrição do script
script_00_create_dirs.sh	Este script remove e cria os diretórios utilizados na execução do GHDL.
script_01_ghdl_import.sh	Este script lista e importa todos os arquivos VHDL que compõem o *testbench* e o *aes_top.vhd*.
script_02_ghdl_make.sh	Este script *analyze* e *elaborate* o *testbench* e o *aes_top.vhd*.
script_03_ghdl_generate_wave.sh	Este script simula o *testbench* e o *aes_top.vhd* e grava a forma de onda em *projectrepository/work/waveform/aes/aes_ctrl_tb/aes_ctrl_tb.vcd*.

O *testbench aes_ctrl_tb* foi simulado duas vezes, isto é, com duas configurações diferentes de instruções e dados seguros. Ao todo foram executados dez testes e todos os testes passaram.

3.3 Implementação e Verificação

Esta seção especifica o desenvolvimento e a verificação do processador LEON3-CONF. A seção está estruturada em três divisões. A Seção 3.3.1 detalha o desenvolvimento do RTL do LEON3-CONF. A Seção 3.3.2 apresenta como a verificação do LEON3-CONF foi realizada. E, por último, a Seção 3.3.3 documenta a síntese do LEON3-CONF.

3.3.1 Desenvolvimento do RTL

O desenvolvimento do RTL do LEON3-CONF significa o projeto do RTL da camada de segurança (Seção 3.1). O desenvolvimento do RTL da camada de segurança foi divido em três etapas e, em cada etapa, o RTL de cada um dos seus blocos foi projetado. A ordem de implementação dos blocos foi:

1. Interface com o barramento AMBA AHB (Seção 3.1.3);
2. Módulo de registradores (Seção 3.1.2);
3. Unidade de encriptação e decriptação (Seção 3.2).

O desenvolvimento de cada um destes blocos compreendeu duas fases de verificação. Na primeira fase, os programas dos *benchmarks* selecionados foram executados no LEON3-CONF simulado com o software GHDL (Apêndice A.1), e as formas de onda geradas foram visualizadas com o software GTKWave (Apêndice A.2). Nesta fase, o *testbench* utilizado é o fornecido com o código fonte do LEON3 (Seção 2.1).

A vantagem desta primeira fase é a visualização dos sinais do LEON3-CONF na forma de onda. Entretanto, o tempo para simular o LEON3-CONF (ou o LEON3) executando um programa em Assembly do *benchmark bench_ 01* (Seção 5.1.1) é aproximadamente 25 minutos, e os programas deste *benchmark* são pequenos (o maior é o *prog_ 11/prog.S* com apenas 122 instruções Assembly no *prog.objdump*).

Na segunda fase da verificação, os programas dos *benchmarks* selecionados foram executados no LEON3-CONF programado em FPGA (Apêndice A.5) e com o auxílio do GRMON2 (Apêndice A.6).

A vantagem desta segunda fase é o tempo necessário para executar um programa (em Assembly ou em linguagem C) no LEON3-CONF (ou no LEON3) programado em FPGA. Este tempo é da ordem de segundos. Apesar do GRMON2 possibilitar o acesso aos registradores, à memória externa e às caches de instrução e de dados, não permite visualizar os sinais do LEON3-CONF (ou do LEON3) como acontece nas formas de onda obtidas na primeira fase.

A segunda fase da verificação inicia-se após o fim da primeira fase, isto é, quando todos os programas dos *benchmarks* selecionados foram executados através da simulação do LEON3-CONF e os resultados dos programas estão de acordo com o esperado.

Na sequência, o desenvolvimento de cada bloco da camada de segurança é detalhado seguindo a ordem de implementação adotada: o primeiro bloco é a interface com o barramento AMBA AHB, o segundo o módulo de registradores, e o terceiro a unidade de encriptação e decriptação.

Interface com o Barramento AMBA AHB

O primeiro bloco da camada de segurança desenvolvido foi a interface com o barramento AMBA AHB (Seção 3.1.3). Para o desenvolvimento deste bloco utilizou-se:

- valores fixos nas entradas a serem geradas pelo módulo de registradores (Seção 3.1.2);
- um módulo "vazio" para a unidade de encriptação e decriptação (Seção 3.2), isto é, os bits encriptados são os próprios bits claros e vice-versa. Além disso, os bits encriptados e decriptados estão sempre disponíveis à interface com o barramento AMBA AHB.

O *benchmark* selecionado para as duas fases da verificação deste bloco foi o *benchmark bench_ 01* (Seção 5.1.1).

Módulo de Registradores

O segundo bloco da camada de segurança desenvolvido foi o módulo de registradores (Seção 3.1.2). Para o seu desenvolvimento utilizou-se a interface com o barramento AMBA AHB (Seção 3.1.3) já implementada e verificada.

Os *benchmarks* selecionados para as duas fases da verificação deste bloco foram os *benchmarks bench_ 01* (Seção 5.1.1) e o *benchmark_ regs* (Seção 5.1.2).

Unidade de Encriptação e Decriptação

O terceiro bloco da camada de segurança desenvolvido foi a unidade de encriptação e decriptação (Seção 3.2).

Primeiro, desenvolveu-se o bloco que implementa o algoritmo AES de encriptação e decriptação (Seção 3.2.2). Para otimizar a verificação, este bloco foi verificado separadamente (Seção 3.2.3).

Depois, a unidade de encriptação e decriptação foi desenvolvida utilizando-se a interface com o barramento AMBA AHB (Seção 3.1.3), o módulo de registradores (Seção 3.1.2) e o algoritmo AES (Seção 3.2.2) implementados e verificados.

O *benchmark* selecionado para as duas fases da verificação da unidade de encriptação e decriptação foi o *benchmark bench_ 01* (Seção 5.1.1).

Neste ponto do projeto, todos os blocos da camada de segurança (Seção 3.1) estavam implementados, portanto o programa (no formato SREC) a ser executado no LEON3-CONF precisava ter as instruções e/ou os dados seguros encriptados (para as duas fases da verificação). Para isso, o programa *srec_ cipher.c* (Seção 5.2.4) foi desenvolvido e utilizado.

Além disso, desenvolveu-se um recurso adicional para a primeira fase da verificação da unidade de encriptação e decriptação (quando o LEON3-CONF é simulado): o monitor de transações AMBA AHB (Seção 3.1.5).

Para verificar os arquivos gerados por este monitor, os arquivos de referência indicados na Tabela 3.20 foram criados para cada programa do *benchmark bench_ 01* e estão no diretório *projectrepository/progs/bench_ 01*.

Tabela 3.20: Arquivos de referência para as transações AMBA AHB

Arquivo de referência	Descrição
ref_ sec_ ahbs_ bus.txt	Este arquivo contém as transações AMBA AHB esperadas para os barramentos seguros *ahbsi* e *ahbso[0]* (Figura 3.1). As instruções e/ou os dados seguros nestes barramentos estão na forma clara.
ref_ nosec_ ahbs_ bus.txt	Este arquivo contém as transações AMBA AHB esperadas para os barramentos não seguros *nosec_ req_ ahbsi* e *nosec_ answer_ ahbso* (Figura 3.1). As instruções e/ou os dados seguros nestes barramentos estão encriptados.

No caso do arquivo de refência *ref_ nosec_ ahbs_ bus.txt*, as instruções e/ou os dados seguros são encriptados pelo programa *aes_ mode.c* (Seção 5.2.3).

3.3.2 Verificação

A verificação do LEON3-CONF foi dividida em duas etapas. Na primeira etapa, cada bloco da camada de segurança (Seção 3.1) foi verificado durante o desenvolvimento do seu RTL (Seção 3.3.1).

Após a conclusão do desenvolvimento do RTL do LEON3-CONF, iniciou-se a segunda etapa da sua verificação. Nesta etapa, os programas dos *benchmarks benchmark_ asm* (Seção 5.1.3) e *benchmark_ c* (Seção 5.1.4) foram executados no LEON3-CONF programado em FPGA (Apêndice A.5) e com o auxílio do GRMON2 (Apêndice A.6).

Estes programas foram executados no LEON3-CONF com sucesso, isto é, todos os resultados obtidos foram iguais aos esperados.

3.3.3 Síntese

O software Intel® Quartus® Prime (Apêndice A.4) foi usado para sintetizar o processador LEON3-CONF e, depois, para programá-lo na plataforma Altera DE2-115 (Apêndice A.5).

A frequência adotada para o clock *clock_ 50* do processador LEON3-CONF (*VHDL entity leon3mp*), que é o mesmo clock *clock* da camada de segurança (*VHDL entity sec_ layer_ top*), foi de 50MHz.

O tempo requerido por este software para sintetizar o processador LEON3-CONF variou entre 1 hora e 30 minutos e 2 horas e 10 minutos.

Capítulo 4

Trabalhos Relacionados

Esta capítulo lista os principais trabalhos relacionados à área desta dissertação, e faz uma análise comparativa com a arquitetura do processador LEON3-CONF. No intuito de consolidar a análise, ao final do capítulo é fornecida a Tabela 4.1 contendo as principais características das soluções propostas pelos artigos analisados.

O artigo *InvisiMem: Smart Memory Defenses for Memory Bus Side Channel*[11] propõe o processador InvisiMem que utiliza *smart memories* (memórias com capacidades computacional) e tecnologia de integração 3D. Neste tipo de tecnologia, camadas de DRAM (*Dynamic Random Access Memory*) são empilhadas sobre uma camada de lógica e são conectadas por *Through Silicon Vias*, que passam através do *silicon wafer*. O InvisiMem também implementa *Intel® Software Guard Extensions*[19] (Intel® SGX): instruções e mecanismos de acessos à região de memória protegida denominada *enclave*. Assim, o InvisiMem garante a confidencialidade e a integridade dos dados nos barramentos processador-memórias e nas memórias externas, além de assegurar proteção ao *timing channel* (quando pretende-se determinar o tempo de acesso e/ou o tempo de resposta da memória a partir do monitoramento do barramento processador-memória).

Apesar do LEON3-CONF utilizar DRAM tradicional, não implementar Intel® SGX e não assegurar a integridade de instruções e de dados, o LEON3-CONF e o InvisiMem adotam a mesma estratégia para garantir a confidencialidade: o processador encripta as instruções e/ou os dados antes de enviá-los à memória externa e os decripta quando os mesmos são recebidos da memória externa. Enquanto no InvisiMem o algoritmo de encriptação e decriptação é o AES em *counter mode*, no LEON3-CONF é o AES.

O método proposto pelo artigo *Improving Cost, Performance, and Security of Memory Encryption and Authentication*[46] garante a integridade e a confidencialidade de instruções e de dados. A integridade é assegurada por *Galois/Counter Mode* (GCM) e *Merkle Tree*, e a confidencialidade por AES em *counter mode* com *split counters*.

No artigo [46], a memória externa é dividida em páginas de encriptação (*encryption pages*), cada página contém 64 blocos e cada bloco 512 bits. No esquema *split counters*, há um *minor counter* (contador de 7 bits) para cada bloco e um *major counter* (contador de 64 bits) para cada página de encriptação. Quando há o *overflow* de um *minor counter*, o *major counter* da página de encriptação correspondente é incrementado e, apenas, esta

página é re-encriptada. Esta é uma das vantagens do modo *split counters* em relação ao uso de apenas um contador, pois, neste último caso, quando ocorre o *overflow* do contador, toda a memória precisa ser re-encriptada.

Nesta implementação do AES em *counter mode*, a chave *key* (Figura 4.1) é gerenciada por um sistema operacional confiável, e a entrada *counter_j* é composta pelo endereço do bloco da memória externa a ser encriptado (ou decriptado), pelos contadores *minor* e *major* deste bloco e pela constante EIV (*Encryption Initialization Vector*).

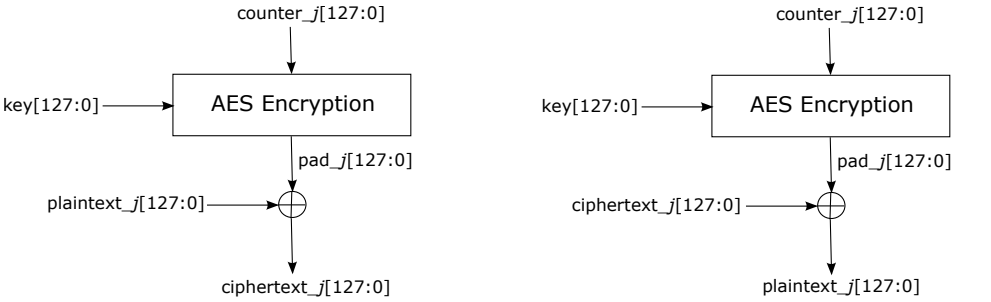

Figura 4.1: AES em *counter mode*.

Os contadores *minor* e *major* são armazenados na memória externa. Por isso, a integridade dos mesmos precisa ser garantida para evitar que eles sejam adulterados e, então, um mesmo *pad_j* seja reusado, por exemplo, através de um *counter replay attack*. A solução adotada pelo artigo foi considerar estes contadores, juntamente com os dados na memória externa, como nós-folhas da *Merkle Tree*.

Os resultados do artigo [46] indicam que o método proposto apresenta uma queda de desempenho de 5% quando comparado a um sistema sem qualquer garantia de integridade e de confidencialidade. Por outro lado, um sistema que utiliza apenas um contador e o algoritmo SHA-1 (*Secure Hash Algorithm 1*), ao invés de *split counters* e GCM, possui uma queda de desempenho de 20%.

O LEON3-CONF e o método do artigo [46] garantem a confidencialidade das instruções e dos dados no barramento processador-memória e na memória externa a partir da encriptação dos mesmos ainda na região segura (Figura 1.1).

O método do artigo [46] garante a integridade das instruções e dos dados, utiliza o AES em *counter mode* com *split counters* para garantir a confidencialidade, e o tamanho dos blocos que são encriptados e decriptados é de 512 bits. Por outro lado, o LEON3-CONF não garante a integridade, utiliza o algoritmo de encriptação e decriptação AES para garantir a confidencialidade, e o tamanho dos blocos que são encriptados e decriptados é de 128 bits.

O artigo *Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly*[39] propõe duas técnicas: *Address Independent Seed Encryption* (AISE) e *Bonsai Merkle Trees* (BMT).

Na técnica AISE (encriptação com semente independente do endereço da memória), as instruções e/ou os dados são encriptados e decriptados através do *counter mode* na região

segura (Figura 1.1) para garantir a confidencialidade dos mesmos. Entretanto, a utilização do endereço (físico ou virtual) da instrução ou do dado na memória como componente da entrada *counter_j* (Figura 4.1) gera problemas de segurança e compromete os mecanismos de memória virtual e de comunicação entre processos (*Inter-Process Communication*).

Por isso, a técnica AISE substitui o endereço da instrução ou do dado por identificadores lógicos na entrada *counter_j*. Assim, a entrada *counter_j* é composta pelo identificador lógico da página LPID (*Logical Page Identifier*), do *offset* do bloco de 128 bits (*chunk*) na página, do contador e do identificador do bloco. O LPID é único por página e só é alterado quando a página precisa ser re-encriptada (devido ao *overflow* do contador do bloco), o *offset* do bloco na página é único para cada bloco em uma determinada página, e o contador do bloco é incrementado toda vez que este bloco é escrito na memória externa.

A integridade das instruções e dos dados é baseada em *Merkle Tree* (MT) que consiste na construção de uma árvore binária onde cada nó armazena um MAC (*Message Authentication Code*). Cada nó-folha da MT armazena o MAC de um bloco da memória externa, assim todos os blocos da memória externa são mapeados por esta árvore. Os demais nós da MT contêm o MAC calculado a partir dos MACs armazenados em seus nós filhos. Desta forma, o nó raiz da MT contém o MAC referente a todos os blocos da memória externa e, por segurança, este MAC é armazenado na região segura (Figura 1.1).

Quando um bloco de dados é escrito pelo processador na memória externa, o MAC deste bloco é calculado, e os MACs, que dependem deste novo MAC, são recalculados e atualizados na MT. Por outro lado, quando o processador lê um bloco de dados da memória externa, o MAC deste bloco é calculado e todos os MACs que dependem deste MAC (inclusive o MAC do nó raiz da MT) são recalculados, e no final, o MAC do nó raiz calculado é comparado com o MAC do nó raiz armazenado na região segura para verificar a integridade do bloco lido. Uma otimização é armazenar os nós da MT recentemente acessados e verificados em uma cache interna à região segura, assim a comparação de um MAC calculado pode ser feita com o MAC correspondente da MT que já foi verificado e que está nesta cache.

Na técnica BMT, os blocos de dados na memória externa são encriptados e decriptados em *counter mode* e um dos componentes da entrada *counter_j* é o contador do bloco. Então, a BMT armazena em cada um dos seus nós-folha o MAC do contador do bloco e não o MAC do bloco de dados (como é feito no caso da MT). Portanto, o tamanho da BMT é menor do que o da MT.

Os resultados do artigo [39] indicam que um sistema com as técnicas AISE e BMT apresenta melhor desempenho e menor área de armazenamento que um sistema com a técnica MT e a encriptação em *counter mode* com um contador global de 64 bits. No primeiro sistema, o aumento médio do tempo de execução é de 1,8% e o aumento total da área de armazenamento é de 21,55%, enquanto que no segundo sistema estes números são 25,9% e 33,51% respectivamente.

O LEON3-CONF e o sistema do artigo [39] garantem a confidencialidade das instruções e dos dados no barramento processador-memória e na memória externa a partir da encriptação dos mesmos ainda na região segura (Figura 1.1).

Contudo, o LEON3-CONF não garante a integridade das instruções e dos dados, uti-

liza o algoritmo AES para garantir a confidencialidade e divide a memória externa em regiões segura (instruções e dados encriptados) e não segura (instruções e dados claros). Já o sistema do artigo [39] garante a integridade das instruções e dos dados, garante a confidencialidade através do algoritmo AES em *counter mode* e não divide a memória externa em regiões segura e não segura.

O artigo *PageVault: Securing Off-Chip Memory using Page-Based Authentication*[45] apresenta o método PageVault que garante a confidencialidade e a integridade dos dados no barramento processador-memória e na memória externa e que reduz a quantidade de metadados utilizados na implementação destas garantias.

A confidencialidade é assegurada por AES em *counter mode* e com *split counters*[46]. A integridade é dividida na integridade dos contadores e na dos blocos de dados. A integridade dos contadores é garantida através de uma *Merkle Tree* (MT) onde cada nó-folha é o valor do contador utilizado para encriptar um bloco de dados e, assim, garantir a confidencialidade deste bloco. A garantia da integridade dos blocos de dados é implementada através da verificação do AMAC (*Aggregation Message Authentication Code*) de cada partição, que é um conjunto de blocos de dados consecutivos na memória externa. O AMAC de uma partição é a combinação do MAC (*Message Authentication Code*) de cada bloco desta partição. Portanto, se um AMAC é verificado, a integridade da partição correspondente e a dos seus blocos são confirmadas. Na implementação do artigo [45], o MAC é calculado através do algoritmo HMAC-SHA-1 (*Hash-based Message Authentication Code - Secure Hash Algorithm 1*), e o AMAC é o *xor* dos MACs dos blocos desta partição.

Os resultados do artigo [45] mostram que quanto maior o número de blocos por partição, menor a quantidade de metadados gerados. Enquanto a estratégia do artigo [39] utiliza 23% da memória externa para armazenar os seus metadados, o esquema PageVault com quatro blocos por partição utiliza apenas 8%. Além disso, o tempo de execução médio dos programas do *benchmark* Splash no esquema PageVault é 10% menor do que na arquitetura do artigo [39] e é 12% menor no caso dos programas do *benchmark* GraphBIG.

O LEON3-CONF e o esquema PageVault[45] garantem a confidencialidade dos dados no barramento processador-memória e na memória externa a partir da encriptação dos dados ainda na região segura (Figura 1.1).

No entanto, o LEON3-CONF não garante a integridade dos dados, utiliza o algoritmo AES para garantir a confidencialidade e divide a memória externa em regiões segura (instruções e dados encriptados) e não segura (instruções e dados claros). O esquema PageVault[45], por sua vez, garante a integridade dos dados, utiliza o algoritmo AES em *counter mode* para garantir a confidencialidade e não divide a memória externa em regiões segura e não segura.

O artigo *CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection*[20] apresenta a arquitetura CryptoPage que garante a confidencialidade e a integridade das instruções e dos dados no barramento processador-memória e na memória externa. Além disso, esta arquitetura protege o barramento de endereços, isto é, impede que os padrões dos endereços da memória externa

acessados pelo processador sejam identificados. Entretanto, nesta arquitetura, o *locked mode* da cache precisa ser utilizado.

Nesta arquitetura, a confidencialidade é garantida pelo algoritmo AES em *counter mode* e a integridade por *Merkle Tree* e pelo MAC (*Message Authentication Code*) de cada linha de bits encriptados (o termo *linha* representa uma linha da memória externa e/ou uma linha da cache). O valor do MAC é calculado pela técnica CBC-MAC (*Cipher Block Chaining Message Authentication Code*).

A arquitetura CryptoPage divide a memória externa em *chunks* e cada *chunk* contém um conjunto de linhas. Se o processador precisa ler uma linha da memória externa (porque esta linha não está na cache) e ela pertence a um *chunk* protegido, então esta linha é lida da memória externa, decriptada e o seu MAC é calculado. Caso o MAC calculado seja igual ao lido da memória externa, a integridade da linha é verificada, logo a linha é utilizada pelo processador, armazenada e *locked* na cache. Enquanto a linha estiver *locked* na cache, ela não é retirada da cache e nem escrita na memória externa.

Se a cache está completa com linhas *locked* e uma nova linha de um *chunk* protegido precisa ser lida da memória externa, então realiza-se uma permutação de *chunk*. Nesta permutação, todas as linhas deste *chunk* protegido (cuja linha será lida) que não estão na cache são lidas da memória externa, decriptadas e têm sua integridade verificada. Depois, os endereços de todas as linhas deste *chunk* são permutados, as linhas que estavam na cache são *unlocked* e re-encriptadas, seu MAC é recalculado e elas são escritas na memória externa. A permutação de *chunk* é utilizada para proteger o barramento de endereços.

Na implementação da arquitetura CryptoPage analisada pelo artigo [20], as instruções protegidas são executadas especulativamente enquanto sua integridade é verificada, as permutações de *chunk* são realizadas em background e iniciadas pelo processador quando este detecta que a cache está quase completa com linhas *locked*. Segundo os resultados deste artigo, a performance desta implementação da arquitetura CryptoPage é 3% menor que a de uma arquitetura sem mecanismos de segurança.

O LEON3-CONF e a arquitetura CryptoPage garantem a confidencialidade das instruções e dos dados no barramento processador-memória e na memória externa através da encriptação dos mesmos. Enquanto o LEON3-CONF utiliza o algoritmo de encriptação e decriptação AES para garantir a confidencialidade, a arquitetura CryptoPage usa o AES em *counter mode*.

Além disso, a política de escrita da cache do LEON3-CONF é *write-through*, então, quando um dado é escrito pelo processador, o mesmo é escrito na cache e na memória externa. Porém, na arquitetura CryptoPage, as linhas de um *chunk* protegido são lidas da memória externa e escritas novamente na memória externa apenas uma vez entre duas permutações deste *chunk*.

O artigo *AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing*[43] apresenta o processador AEGIS que garante a confidencialidade e a integridade de instruções e de dados. A arquitetura do AEGIS utiliza o algoritmo AES em CBC (*Cipher Block Chaining*) *mode* para assegurar a confidencialidade, *Merkle Trees* e gerenciador de contextos seguro (*secure context manager*) para assegurar a integridade. O AEGIS também inclui uma camada de software que depende do sistema operacional

utilizado. Esta camada contém mais funcionalidades quando o sistema operacional não é seguro (*untrusted operating system*) e menos funcionalidades quando o sistema operacional possui uma parte segura denominada *security kernel*.

De acordo com os resultados do artigo [43], a garantia de confidencialidade e de integridade de todas as instruções e de todos os dados reduzem o desempenho do processador AEGIS, na maioria dos casos, em 40% e, no pior caso, em 60%.

O LEON3-CONF e o processador AEGIS garantem a confidencialidade das instruções e dos dados no barramento processador-memória e na memória externa. E, nos dois casos, a memória externa é dividida em regiões segura e não segura.

Porém, o LEON3-CONF garante a confidencialidade através do algoritmo de encriptação e decriptação AES e não garante a integridade. Já o processador AEGIS garante a confidencialidade através do algoritmo AES em CBC *mode* e garante a integridade.

O artigo *Design of A Pre-Scheduled Data Bus for Advanced Encryption Standard Encrypted System-on-Chips*[48] propõe o barramento de dados (*data bus*) DBUS, que é uma extensão do barramento MSBUS (*master-slave bus*) definido no artigo *A High-Performance On-Chip Bus (MSBUS) Design and Verification*[47]. Além de suportar as transferências linear e de bloco do barramento MSBUS, o DBUS suporta as transferências de estado (*state transfer*). A unidade básica das transferências de estado é uma matriz 4x4 de bytes organizados de acordo com a etapa *shift rows* (do algoritmo AES de encriptação) ou com a etapa *inverse shift rows* (do algoritmo AES de decriptação), e estes bytes são a entrada do próximo *round* do algoritmo AES de encriptação ou de decriptação.

Na arquitetura utilizada pelo artigo [48] (Figura 4.2), quando um *master* faz um pedido de leitura ao DMA (*Direct Memory Access*), o dado lido da memória interna é encriptado antes de ser enviado ao barramento DBUS (ou ao barramento AXI[12]). E, quando um *master* faz um pedido de escrita ao DMA, o dado é decriptado antes de ser escrito na memória interna. Os resultados deste artigo demonstram que a arquitetura com o barramento DBUS apresenta menor área, maior frequência de operação, maior taxa de transferência (*throughput*) e menor consumo de potência dinâmica quando comparada à arquitetura com o barramento AXI.

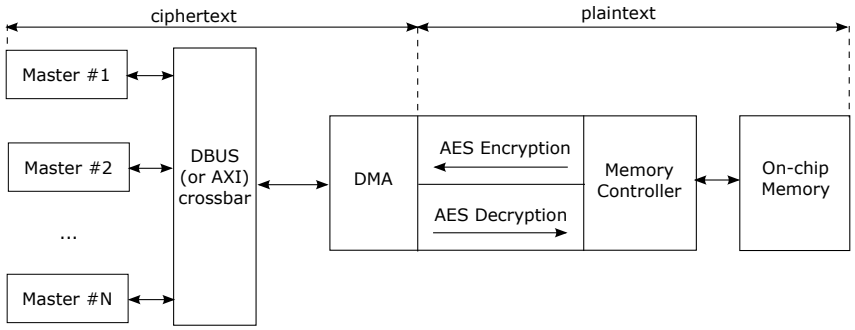

Figura 4.2: Arquitetura utilizada pelo artigo [48].

O LEON3-CONF e o artigo [48] utilizam o algoritmo AES para garantir a confidencialidade em seus barramentos. O LEON3-CONF garante a confidencialidade em um barramento padrão: AMBA (*Advanced Microcontroller Bus Architecture*) AHB

(*Advanced High-performance Bus*)[34], e este artigo em um novo tipo de barramento: DBUS.

O LEON3-CONF também garante a confidencialidade das instruções e dos dados armazenados na memória externa. Entretanto, no caso de estudo do artigo [48], os dados são armazenados na forma clara em uma memória interna ao SoC (*System-on-Chip*) e acessada apenas pelo DMA.

A principal proposta do artigo *DEUCE: Write-Efficient Encryption for Non-Volatile Memories*[49] é a arquitetura DEUCE (*Dual Counter Encryption*) que garante a confidencialidade dos dados no barramento processador-memória e na memória externa. Diferente dos demais artigos analisados e da arquitetura do LEON3-CONF, a memória externa no artigo [49] é uma memória não volátil (*Non Volatile Memory*) *Phase Change Memory* (PCM). No caso destas memórias, quanto maior o número de bits escritos, maior o seu consumo de potência, menor o seu desempenho e o seu tempo de vida. Por isso, a arquitetura DEUCE também otimiza o número de bits encriptados escritos na PCM.

A arquitetura DEUCE utiliza o algoritmo AES em *counter mode*, um contador LCTR (*Leading Counter*) para cada linha da cache que é incrementado quando há uma escrita nesta linha, um contador TCTR (*Trailing Counter*) para cada contador LCTR (o TCTR é o LCTR com os 5 bits menos significativos deste último contador iguais a zero), um bit para cada *word* na linha da cache para indicar se a *word* foi escrita ou não, e cada *word* tem 2 bytes. Se a *word* não foi escrita, então ela é encriptada com o pad_j (Figura 4.1) obtido com TCTR (valor do contador usado para encriptar a linha inicialmente), logo esta *word* permanece inalterada. Mas, se ela foi escrita, então ela é encriptada com o pad_j determinado pelo LCTR (o contador atualizado), portanto ela é modificada. A cada 32 escritas na mesma linha da cache, os 32 bits de controle são zerados, LCTR e TCTR tornam-se iguais, e a linha da cache é re-encriptada usando o pad_j gerado pelo novo valor destes contadores.

Os resultados do artigo [49] indicam que a arquitetura DEUCE reduz de 50% para 24% o número de bits de uma linha da cache (depois que a linha foi encriptada) escritos na PCM, diminui o consumo de potência da PCM em 28%, aumenta o desempenho da PCM em 27% e o seu tempo médio de vida em 11%.

O LEON3-CONF e a arquitetura DEUCE garantem a confidencialidade dos dados no barramento processador-memória e na memória externa através da encriptação dos dados antes deles saírem da região segura (Figura 1.1). Em ambos, cada bloco de bits encriptados e escritos na memória externa (e lidos da memória externa e decriptados) é do tamanho de uma linha da cache usada. Além disso, as duas estratégias não garantem a integridade dos dados.

Enquanto na arquitetura DEUCE, a política de escrita da cache é *write-back* (os dados escritos na cache são apenas escritos na memória principal de tempos em tempos ou em determinada condição), o tamanho da linha da cache é de 512 bits, a memória externa é uma PCM, o número de bits encriptados escritos na memória externa é otimizado, e o algoritmo de encriptação e decriptação é o AES em *counter mode*, no LEON3-CONF, a política de escrita da cache é *write-through*, o tamanho da linha da cache é de 128 bits, a memória externa é uma DRAM, o número de bits encriptados escritos na memória externa

não é otimizado, e o algoritmo de encriptação e decriptação é o AES.

O artigo *Protecting Data on Smartphones and Tablets from Memory Attacks*[16] apresenta o sistema Sentry para proteção de dados sigilosos em dispositivos móveis (*tablets* e *smartphones*) de ataques dos tipos *cold boot*, *bus monitoring* e DMA. O sistema Sentry visa proteger o conteúdo da memória externa apenas quando a tela do dispositivo está bloqueada, pois quando a mesma não está bloqueada, o atacante tem acesso direto às aplicações e aos dados através da interface do usuário. No estado de tela bloqueada, o dispositivo pode executar ou não aplicações em background.

A implementação do sistema Sentry é baseada em dois mecanismos da arquitetura ARM (inicialmente projetados para aplicações de tempo real): RAM interna (iRAM) ao SoC (*System-on-Chip*) e *Locked L2 Cache*. Na versão completa deste sistema, os dados sigilosos são armazenados na iRAM ou em regiões seguras da memória externa. Neste segundo caso, as páginas da cache L2 (referente às regiões seguras da memória externa) são encriptadas no SoC antes de serem enviadas à memória externa. E, quando estas páginas são lidas da memória externa, as mesmas são decriptadas no SoC e, só então, enviadas à cache L2.

O LEON3-CONF e o sistema Sentry garantem a confidencialidade dos dados no barramento processador-memória e na memória externa. Além disso, ambos dividem a memória externa em regiões segura e não segura.

Contudo, o sistema Sentry implementa a encriptação e a decriptação dos dados sigilosos em software usando o algoritmo AES em CBC (*Cipher Block Chaining*) *mode* e sua unidade de encriptação e decriptação é uma página da cache L2 (blocos de 4 KB). Por outro lado, o LEON3-CONF implementa em hardware o algoritmo AES e a sua unidade de encriptação e decriptação são 128 bytes (uma linha da cache de instruções e da de dados). Além destas diferenças, o sistema Sentry depende da arquitetura ARM, protege dados sigilosos apenas em dispositivos móveis bloqueados, mas é capaz de proteger dados sigilosos de ataques do tipo DMA.

O artigo *Operating System Controlled Processor-Memory Bus Encryption*[15] garante a confidencialidade dos dados no barramento processador-memória e na memória externa exclusivamente através de software. Este artigo propõe um sistema operacional que encripta os dados antes deles saírem da região segura (Figura 1.1) e os decripta quando eles chegam à região segura. A única restrição é que o sistema operacional precisa controlar as transferências de dados entre a cache e a memória externa, portanto o processador deve suportar cache com *locked mode* ou uma cache que possa ser usada como uma memória endereçável (*scratchpad memory*).

Neste artigo, o termo "página segura" (*secure page*) representa um bloco da memória externa que é mapeado em uma única página da cache e que pertence a um processo protegido. Quando o sistema operacional proposto lê uma página segura da memória externa, ele a decripta e, então, a escreve na cache (na forma clara). E, quando ele precisa escrever na memória externa uma página segura que está na cache, ele a encripta e, depois, a escreve na memória externa (na forma encriptada).

Outra particularidade deste sistema operacional está nas trocas de contexto. Quando

o novo contexto é um contexto de um processo protegido, o sistema operacional assume o controle das transferências de dados entre a cache e a memória externa (habilita o *locked mode* da cache ou utiliza a cache como uma memória endereçável). E, quando a troca de contexto é de um processo protegido para um processo não protegido, o sistema operacional invalida o conteúdo da cache referente ao processo anterior para evitar que o novo processo tenha acesso aos dados do anterior.

De acordo com os resultados do artigo [15], o desempenho do sistema operacional proposto quando processos protegidos são executados é diretamente proporcional ao tamanho da cache e inversamente proporcional ao tamanho das páginas da cache. E, quando processos não protegidos são executados, as penalidades no desempenho deste sistema operacional são insignificantes.

O LEON3-CONF e o sistema operacional do artigo [15] garantem a confidencialidade dos dados no barramento processador-memória e na memória externa. Ambos não garantem a integridade de dados, e o desempenho das duas implementações não é alterado quando processos não protegidos (no caso deste artigo) ou programas com instruções e dados não seguros (no caso do LEON3-CONF) são executados.

Entretanto, o LEON3-CONF garante a confidencialidade através da implementação em hardware da camada de segurança (Seção 3.1), e a solução do artigo [15] garante a confidencialidade através de software (sistema operacional). Além disso, neste artigo, o processador precisa suportar cache com *locked mode* (ou *scratchpad memory* para ser usada como cache), o algoritmo de encriptação e decriptação é o algoritmo AES em CBC (*Cipher Block Chaining*) *mode*, o bloco de dados enviados pelo processador à memória externa (e vice-versa) é uma página da cache, e, no caso de um processo protegido, todos os seus dados são protegidos. Por outro lado, no LEON3-CONF, o algoritmo de encriptação e decriptação é o AES, o bloco de dados enviados pelo processador à memória externa (e vice-versa) é uma linha da cache (128 bits), e um programa pode ter parte de suas instruções e/ou de seus dados protegidos.

Tabela 4.1: Artigos analisados

Solução	Confidencialidade	Integridade	Observações
Processador *InvisiMem*[11]	AES em *counter mode*	*smart memories*, integração 3D e Intel® SGX	proteção ao *timing channel*
Método do artigo [46]	AES em *counter mode* com *split counters*	GCM e *Merkle Tree*	sistema operacional confiável
Sistema do artigo [39]	AES em *counter mode* com a técnica AISE	BTM	-
Método PageVault[45]	AES em *counter mode* com *split counters*	*Merkle Tree*, HMAC-SHA-1 e AMAC	-
Arquitetura CryptoPage[20]	AES em *counter mode*	*Merkle Tree*, CBC-MAC	cache com *locked mode*, permutação de *chunk* e proteção do barramento de endereço
Processador AEGIS[43]	AES em CBC *mode*	*Merkle Tree* e *secure context manager*	camada de software que depende do sistema operacional
Barramento DBUS[48]	AES	não garante a integridade	os dados são armazenados na forma clara em uma memória interna acessada apenas pelo DMA
Arquitetura DEUCE[49]	AES em *counter mode*	não garante a integridade	a memória externa é PCM e o número de bits encriptados escritos na PCM é otimizado
Sistema Sentry[16]	AES em CBC *mode* (em software)	não garante a integridade	RAM interna (iRAM) e *Locked L2 Cache*
Sistema operacional do artigo [15]	AES em CBC *mode* (em software)	não garante a integridade	cache com *locked mode*

Capítulo 5

Resultados Experimentais

Este capítulo descreve os resultados experimentais obtidos quando o processador LEON3-CONF executa um conjunto de programas de teste. O capítulo está dividido em cinco seções. A Seção 5.1 lista os *benchmarks* desenvolvidos para verificar e avaliar a arquitetura proposta. A Seção 5.2 apresenta os programas desenvolvidos para realizar a encriptação AES, por exemplo, para encriptar as instruções e/ou os dados seguros do programa a ser executado no LEON3-CONF. A Seção 5.3, por sua vez, detalha como os segmentos de memória para instruções seguras e para dados seguros são especificados e utilizados por programas em linguagem C executados no LEON3-CONF. A Seção 5.4 descreve o impacto no tempo de execução dos programas do *benchmark benchmark_ c* executados no LEON3-CONF, enquanto a Seção 5.5 apresenta o custo em termos de área da FPGA necessário à implementação da camada de segurança do processador LEON3-CONF.

5.1 *Benchmarks*

Esta seção documenta os *benchmarks* desenvolvidos para serem executados, principalmente, no LEON3-CONF com o objetivo de verificar a sua funcionalidade e avaliar o seu desempenho. Esta seção está estruturada em quatro divisões. A Seção 5.1.1 apresenta o *benchmark bench_ 01* e os seus três programas escritos em Assembly. Os programas deste *benchmark* foram executados no LEON3 e no LEON3-CONF através da simulação do processador ou quando o processador está programado em FPGA. A Seção 5.1.2 detalha o *benchmark benchmark_ regs* e os seus dois programas escritos em Assembly. Os dois programas foram desenvolvidos para serem executados no LEON3-CONF, o primeiro programa através da simulação do processador e o segundo quando o processador está programado em FPGA. A Seção 5.1.3 especifica o *benchmark benchmark_ asm* e os seus quatro programas escritos em Assembly. Estes programas foram executados no LEON3-CONF programado em FPGA e com o auxílio do GRMON2 para verificar o LEON3-CONF. Por último, a Seção 5.1.4 descreve o *benchmark benchmark_ c* e os seus cinco programas escritos em linguagem C. Os programas deste *benchmark* foram executados no LEON3-CONF programado em FPGA e com o auxílio do GRMON2 para verificar o LEON3-CONF e avaliar o seu desempenho.

5.1.1 *Benchmark bench_ 01*

O *benchmark bench_ 01* é composto por três programas escritos em Assembly e executados no LEON3 e no LEON3-CONF através da simulação do processador ou quando o processador está programado em FPGA. Estes programas foram executados no LEON3 para permitir a análise da arquitetura e do funcionamento deste processador. No caso do LEON3-CONF, estes programas foram utilizados durante o desenvolvimento da camada de segurança (Seção 3.1). A Tabela 5.1 contém os programas deste *benchmark*.

Tabela 5.1: Programas do *benchmark bench_ 01*

Nome do programa	Descrição do programa
prog_01/prog.S	O programa escreve valores em quatro posições da memória e lê estas quatro posições da memória.
prog_10/prog.S	O programa lê quatro posições da memória.
prog_11/prog.S	O programa escreve em oito posições da memória e depois lê as quatro posições que foram escritas primeiro.

No LEON3-CONF, cada um destes programas foi executado nas quatro configurações:

- instruções e dados não seguros;
- instruções não seguras e dados seguros;
- instruções seguras e dados não seguros;
- instrucões e dados seguros.

Os programas da Tabela 5.1 foram escolhidos para serem executados no LEON3-CONF, porque a execução destes três programas nas quatro configurações exercitam todos os estados e arcos da máquina de estados implementada na interface com o barramento AMBA AHB (Seção 3.1.3).

Esta análise dos estados e arcos exercitados foi realizada a partir do valor armazenado no registrador SEC_PERF_FSM_CP_REG (Tabela 3.9) ao final da execução de cada programa e em cada configuração.

Além disso, os arquivos de referência com o valor esperado para as posições da memória utilizadas (Tabela 5.2) foram feitos para facilitar a comparação do conteúdo esperado com o obtido na execução de cada programa no LEON3-CONF programado em FPGA.

O programa *aes_ mode.c* (Seção 5.2.3) foi utilizado para encriptar as instruções e/ou os dados seguros para cada arquivo de referência.

Todos os programas deste *benchmark* e os arquivos de referência (para cada programa) estão no diretório *projectrepository/progs/bench_ 01*.

5.1.2 *Benchmark benchmark_ regs*

O *benchmark benchmark_ regs* contém dois programas escritos em Assembly para verificar os registradores implementados no módulo de registradores (Seção 3.1.2).

Tabela 5.2: Arquivos de referência para as posições da memória

Arquivo de referência	Descrição
ref_inst_clara.txt	Posições da memória com as instruções do programa a ser executado. As instruções seguras estão na forma clara.
ref_inst_cif.txt	Posições da memória com as instruções do programa a ser executado. As instruções seguras estão encriptadas.
ref_dado_cif_antes_rodar.txt	Posições da memória com os dados iniciais do programa a ser executado. Os dados seguros estão encriptados.
ref_dado_cif_depois_rodar.txt	Posições da memória com os dados após a execução do programa. Os dados seguros estão encriptados.
ref_dado_claro_depois_rodar.txt	Posições da memória com os dados após a execução do programa. Os dados seguros estão na forma clara.

Os programas deste *benchmark* estão no diretório *projectrepository/progs/ benchmark_regs*.

O programa *prog_01/prog.S* escreve um valor em cada registrador do módulo de registradores e, em seguida, lê este registrador. O objetivo foi desenvolver um programa simples e com o menor tempo de execução possível, assim este programa pôde ser executado no LEON3-CONF através de sua simulação juntamente com o seu *testbench*. Os valores escritos nos registradores foram conferidos na forma de onda gerada pela simulação.

O programa *prog_02/prog.S* tem mais funções que o anterior, porque o programa *prog_02* foi desenvolvido para ser executado quando o LEON3-CONF está programado em FPGA, portanto o seu tempo de execução pôde ser maior. Resumidamente, para cada registrador, o programa *prog_02*:

1. Escreve um valor no registrador;
2. Lê o registrador escrito;
3. Escreve o valor lido em uma variável (posição da memória).

No caso do programa *prog_02*, a verificação dos valores escritos nas variáveis foi feita com o auxílio do GRMON2 (Apêndice A.6).

5.1.3 *Benchmark benchmark_asm*

O *benchmark benchmark_asm* é o conjunto de programas escritos em Assembly executados no LEON3-CONF programado em FPGA e com o auxílio do GRMON2 (Apêndice A.6).

A Tabela 5.3 lista os programas deste *benchmark*. Cada um destes programas foi executado em quatro configurações diferentes de acordo com a Tabela 5.4.

Tabela 5.3: Programas em Assembly do *benchmark benchmark_ asm*

Nome do programa	Descrição do programa
prog_*1/prog.S	O programa escreve valores em quatro posições da memória e lê estas quatro posições da memória.
prog_*2/prog.S	O programa lê quatro posições de memória, executa cinco operações (adição, *and* lógico, subtração, *or* lógico e *xor* lógico) e os resultados são escritos em outras cinco posições da memória.
prog_*5/prog.S	O programa ordena o conteúdo de 20 posições da memória através do algoritmo de ordenação *Bubble Sort*.
prog_*6/prog.S	O programa multiplica dois números (sem considerar o sinal deles) armazenados em duas posições da memória, e escreve o resultado em outra posição da memória.

Tabela 5.4: Configurações dos programas do *benchmark benchmark_ asm*

Nome do programa	Configuração do programa
prog_0*/prog.S	Programa com instruções e dados não seguros
prog_1*/prog.S	Programa com instruções não seguras e dados seguros
prog_2*/prog.S	Programa com instruções seguras e dados não seguros
prog_3*/prog.S	Programa com instruções e dados seguros

Da mesma forma que no *benchmark bench_ 01* (Seção 5.1.1), os arquivos de referência (Tabela 5.2) foram desenvolvidos.

Todos os programas deste *benchmark* e os arquivos de referência (para cada programa) estão no diretório *projectrepository/progs/benchmark_ asm*.

5.1.4 *Benchmark benchmark_ c*

O *benchmark benchmark_ c* contém os programas escritos em linguagem C utilizados para verificar o LEON3-CONF e avaliar o seu desempenho.

Os programas deste *benchmark* (Tabela 5.5) foram executados no LEON3-CONF programado em FPGA e com o auxílio do GRMON2 (Apêndice A.6). Cada programa foi executado em quatro configurações diferentes de acordo com a Tabela 5.6.

Os dados seguros de cada um dos programas deste *benchmark* (Tabela 5.7) são armazenados no segmento de dados seguros *secdata* (Seção 5.3), e as instruções seguras (Tabela 5.8) no segmento de instruções seguras *sectext* (Seção 5.3).

Da mesma forma que no *benchmark bench_ 01* (Seção 5.1.1), os arquivos de referência (Tabela 5.2) foram desenvolvidos.

Tabela 5.5: Programas em linguagem C do *benchmark benchmark_ c*

Nome do programa	Descrição do programa
prog_01_*/prog.c	O programa executa cinco operações (adição, subtração, multiplicação, divisão inteira e resto da divisão inteira) com dois operandos e armazena o resultado de cada operação.
prog_02_*/prog.c	O programa ordena um vetor com 20 posições através do algoritmo de ordenação *Bubble Sort*.
prog_03_*/prog.c	O programa calcula o valor da sequência de Fibonacci F(n) (para *n=20*) utilizando recursão.
prog_04_*/prog.c	O programa calcula o valor da sequência de Fibonacci F(n) (para *n=20*) utilizando recursão.
prog_05_*/prog.c	O programa cria uma lista ligada, copia os elementos desta lista em um vetor e libera as posições de memória alocadas dinamicamente.

Tabela 5.6: Configurações dos programas do *benchmark benchmark_ c*

Nome do programa	Configuração do programa
prog_0*_00/prog.c	Programa com instruções e dados não seguros
prog_0*_01/prog.c	Programa com instruções não seguras e dados seguros
prog_0*_10/prog.c	Programa com instruções seguras e dados não seguros
prog_0*_11/prog.c	Programa com instruções e dados seguros

Todos os programas deste *benchmark* e os arquivos de referência (para cada programa) estão no diretório *projectrepository/progs/benchmark_ c*.

5.2 Programas Desenvolvidos

Esta seção detalha a função e os programas implementados para realizar a encriptação AES ou para auxiliar na análise dos resultados dos programas executados no LEON3-CONF ou para encriptar as instruções e/ou os dados seguros do programa a ser executado no LEON3-CONF. Esta seção está estruturada em quatro divisões. A Seção 5.2.1 documenta a função *aes_ cipher_ func* desenvolvida para encriptar 128 bits usando o algoritmo AES de encriptação. A Seção 5.2.2 apresenta o programa *aes_ cipher_ tb.c* desenvolvido para verificar a função *aes_ cipher_ func*. A Seção 5.2.3 especifica o programa *aes_ mode.c* desenvolvido para encriptar (através da função *aes_ cipher_ func*) apenas 128 bits e auxiliar na análise dos resultados dos programas executados no LEON3-CONF. Por último, a Seção 5.2.4 descreve o programa *srec_ cipher.c* desenvolvido para encriptar (através da função *aes_ cipher_ func*) as instruções e/ou os dados seguros de um programa no formato SREC a ser executado no LEON3-CONF.

Tabela 5.7: Dados seguros dos programas do *benchmark benchmark_ c*

Nome do programa	Dados seguros
prog_01_*/prog.c	Os dois operandos (*value_ 1* e *value_ 2*) e os cinco resultados (*result_ 1*, *result_ 2*, *result_ 3*, *result_ 4* e *result_ 5*).
prog_02_*/prog.c	O vetor *vector* e a variável *temp_ value* utilizada para armazenar temporariamente uma posição do vetor.
prog_03_*/prog.c	O valor de *n* (*value_ n*) e o resultado da sequência de Fibonacci F(n) (*result*).
prog_04_*/prog.c	O valor de *n* (*value_ n*), os valores intermediários e o resultado da sequência de Fibonacci F(n) (vetores *rec_ res_ 0* e *rec_ res_ 1*).
prog_05_*/prog.c	Os vetores *vector_ 1*, *vector_ 2* e *vector_ 3*.

Tabela 5.8: Instruções seguras dos programas do *benchmark benchmark_ c*

Nome do programa	Instruções seguras
prog_01_*/prog.c	As instruções na função *operations*.
prog_02_*/prog.c	As instruções na função *sort_ vector*.
prog_03_*/prog.c	As instruções nas funções *func_ calls_ secfunc* e *fibonacci*.
prog_04_*/prog.c	As instruções nas funções *func_ calls_ secfunc* e *fibonacci*.
prog_05_*/prog.c	As instruções nas funções *func_ calls_ secfunc*, *add_ node*, *create_ list*, *write_ list*, *free_ list* e *secfunc*.

5.2.1 Função *aes_ cipher_ func*

A função *aes_ cipher_ func* foi desenvolvida em linguagem C e o seu código fonte está em *projectrepository/tool_ data/cipher/source/aes_ cipher.c*.

Os parâmetros de entrada desta função são: os 128 bits a serem encriptados (*plain_ bits[127:0]*) e a chave de 128 bits (*in_ key[127:0]*). A função retorna os 128 bits encriptados pelo algoritmo AES.

5.2.2 Verificação da Função *aes_ cipher_ func*

O programa *aes_ cipher_ tb.c* (Figura 5.1) foi desenvolvido em linguagem C para verificar a funcionalidade da função *aes_ cipher_ func*.

O programa *aes_ cipher_ tb.c* está em *projectrepository/tool_ data/cipher/testbench*.

O programa *aes_ cipher_ tb.c* foi executado com dois conjuntos de testes. Cada teste contém uma chave de 128 bits, 128 bits claros e 128 bits encriptados. O programa *aes_ cipher_ tb.c* lê do arquivo de entrada *input_ file_ name* estes dados, executa a função *aes_ cipher_ func* utilizando a chave e os bits claros lidos, compara o resultado desta função com os bits encriptados lidos, e escreve no arquivo de saída *output_ file_ name* o

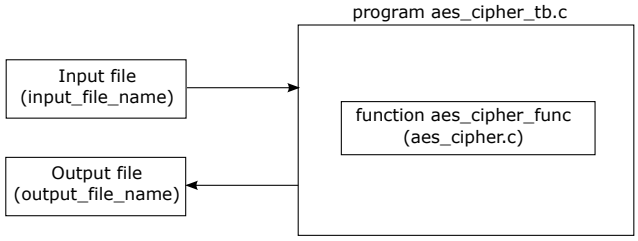

Figura 5.1: Programa *aes_cipher_tb.c*.

resultado desta comparação.

Os arquivos de entrada com os testes a serem executados estão em *projectrepository/tool_data/cipher/stimulus*, e os arquivos de saída com os resultados da encriptação AES estão em *projectrepository/tool_data/cipher/result*.

O primeiro conjunto de testes está no arquivo de entrada *aes_3_input_tests.txt*. Este conjunto de testes contém 3 testes. Os dois primeiros testes são da referência [21] e o terceiro é do arquivo *ecb_tbl.txt* (*rijndael-vals.zip*[37]). O resultado desta execução do programa *aes_cipher_tb.c* está no arquivo de saída *aes_3_output_tests.txt*. Os três testes passaram.

O segundo conjunto de testes está no arquivo de entrada *aes_nist_input_tests.txt* que contém os 128 testes do arquivo *ecb_tbl.txt* (*rijndael-vals.zip*[37]). Este é o mesmo conjunto de testes utilizado no Seção 3.2.3. O resultado desta execução do programa *aes_cipher_tb.c* está no arquivo de saída *aes_nist_output_tests.txt*. Os 128 testes passaram.

5.2.3 Programa *aes_mode.c*

O programa *aes_mode.c* (Figura 5.2) foi desenvolvido para realizar a encriptação AES de 128 bits e para auxiliar na análise das instruções e/ou dos dados seguros dos programas executados no LEON3-CONF.

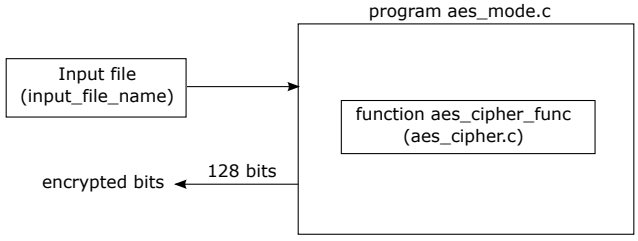

Figura 5.2: Programa *aes_mode.c*.

O programa *aes_mode.c* lê do arquivo de entrada *input_file_name* a chave de 128 bits e os 128 bits claros, chama a função *aes_cipher_func* com os dados lidos e escreve na tela os 128 bits encriptados.

O código fonte do programa *aes_mode.c* está em *projectrepository/tool_data/cipher_line/aes_mode/source*, e o arquivo de entrada *input_file.txt* em *projectrepository/tool_data/cipher_line/aes_mode/input_files*.

5.2.4 Programa *srec_ cipher.c*

O programa *srec_ cipher.c* foi desenvolvido em linguagem C para encriptar, através do algoritmo de encriptação AES, as instruções e/ou os dados seguros de um programa no formato SREC[44]. Este programa também utiliza a função *aes_ cipher_ func*.

A Figura 5.3 apresenta o fluxo utilizado para gerar um programa no formato SREC com instruções e/ou dados encriptados a partir de um programa em Assembly ou em linguagem C.

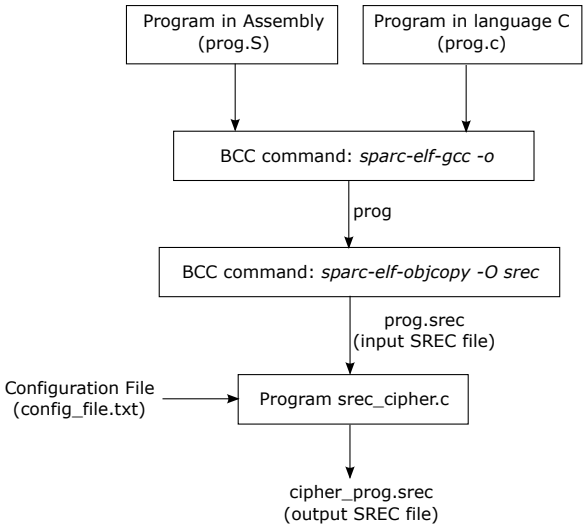

Figura 5.3: Geração de um programa com instruções e/ou dados encriptados.

O programa *srec_ cipher.c* (Figuras 5.3 e 5.4) utiliza o arquivo de configuração *config_ file.txt* e o programa de entrada *prog.srec* (no formato SREC) para gerar o programa *cipher_ prog.srec* com as instruções e/ou dados seguros encriptados (no formato SREC).

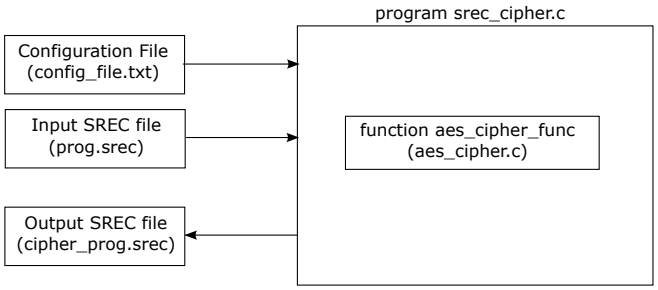

Figura 5.4: Programa *srec_ cipher.c*.

O programa de entrada (no formato SREC) é gerado pelo BCC (Apêndice A.3) a partir de um programa escrito em Assembly ou em linguagem C.

A Tabela 5.9 contém os campos do arquivo de configuração *config_ file.txt*, e a Tabela 5.10 lista a localização do código fonte do programa *srec_ cipher.c* e a dos arquivos de entrada e de saída utilizados pelo programa *srec_ cipher.c*.

5.3 Segmentos de Memória Segura

Para os programas do *benchmark benchmark_ c* (Seção 5.1.4) criou-se um segmento de memória para as instruções seguras *sectext* e um segmento de memória para os dados seguros *secdata* (Figura 5.5).

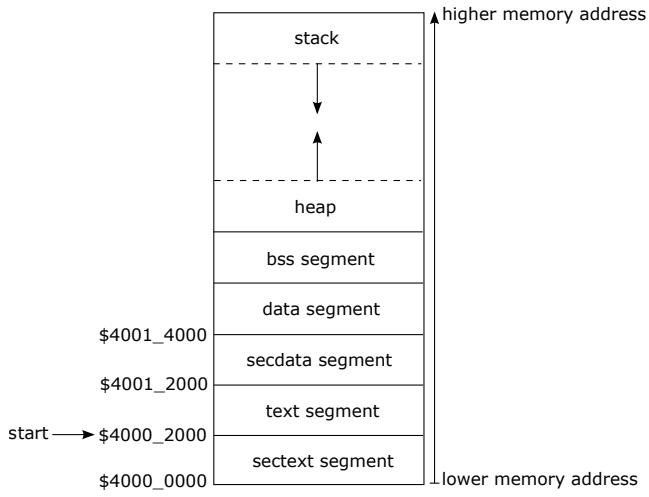

Figura 5.5: Layout da memória para programas em linguagem C no LEON3-CONF.

A criação dos segmentos *sectext* e *secdata* é realizada pelo *linker*[14, 38] (última etapa da compilação de um programa em linguagem C) através do *linker script*.

O *linker script* do LEON3 é *sparc-elf/lib/ldscripts/sparcleon.x* e está no diretório de instalação do BCC (Apêndice A.3). O *linker script* do LEON3-CONF é *sparcsecleon.x* e está em *projectrepository/progs/benchmark_ c/c_files*.

Entretanto, em cada programa em linguagem C, é necessário indicar ao compilador e ao *linker* quais são as instruções e/ou os dados seguros através da diretiva de compilação *__attribute__*[42].

Para simplificar o uso desta diretiva de compilação, o arquivo *projectrepository/progs/benchmark_ c/c_files/sec_ layer.h* contém os *defines* *SEC_ LAYER_ SEC_ INST* e *SEC_ LAYER_ SEC_ DATA*.

O *define SEC_ LAYER_ SEC_ INST* é utilizado para indicar que as instruções de uma função são seguras e, por isso, devem ser armazenadas no segmento de memória de instruções seguras *sectext*. A declaração deste *define* é exibida a seguir.

```
// Secutiry layer - security instruction
#define SEC_LAYER_SEC_INST __attribute__((section(".sectext")))
```

Por outro lado, o *define SEC_ LAYER_ SEC_ DATA* é usado para indicar quando as variáveis são seguras e, portanto, devem ser armazenadas no segmento de memória de dados seguros *secdata*. A declaração deste *define* é mostrada na sequência.

```
// Secutiry layer - security data
#define SEC_LAYER_SEC_DATA __attribute__((section(".secdata")))
```

5.4 Avaliação de Desempenho

O desempenho do processador LEON3-CONF foi determinado a partir do número de períodos de clock (como descrito na Seção 3.1.4) usados na execução de cada configuração dos programas do *benchmark benchmark_ c* (Seção 5.1.4).

A Tabela 5.11 mostra os resultados obtidos. A segunda coluna desta tabela ("Instruções e dados não seguros") indica o número total de períodos de clock utilizados na execução de cada programa com instruções e dados não seguros, isto é, o valor de referência para a avaliação de desempenho do LEON3-CONF. Por outro lado, as demais colunas da Tabela 5.11 ("Instruções não seguras e dados seguros", "Instruções seguras e dados não seguros" e "Instruções e dados seguros") indicam o número de vezes em relação ao número de períodos de clock da segunda coluna.

5.5 Avaliação de Área

A análise da área do processador LEON3-CONF foi realizada a partir dos resultados da sua síntese (Seção 3.3.3).

O relatório de síntese utilizado para esta análise é o *leon3mp.map.rpt*[18] e está no diretório *projectrepository/work/quartus/leon3_ sec*.

A seção "*Analysis & Synthesis Resource Utilization by Entity*" deste relatório contém o número de células lógicas (*logic cell*) combinacionais (*LC Combinationals*) e de registradores (*LC Registers*) para cada bloco do LEON3-CONF sintetizado.

O termo "célula lógica"[17] é um termo genérico adotado pelo software Intel® Quartus® Prime (Apêndice A.4) para indicar um bloco básico de construção. Para simplificar as comparações, as Tabelas 5.12, 5.13 e 5.14 utilizam a porcentagem de lógica combinacional (ao invés do número de células combinacionais) e a porcentagem de registradores (no lugar do número de registradores).

A Tabela 5.12 exibe a porcentagem da área do LEON3-CONF em relação à área do LEON3, a Tabela 5.13 destaca a porcentagem da área da camada de segurança (Seção 3.1) em relação à área do LEON3-CONF, e a Tabela 5.14 detalha a porcentagem da área de cada bloco da camada de segurança em relação à área total da camada de segurança.

Tabela 5.9: Campos do arquivo de configuração *config_file.txt*

Campo de configuração	Descrição
Input SREC file	Nome e caminho do programa de entrada no formato SREC.
Output SREC file	Nome e caminho do programa de saída no formato SREC.
SEC_DATA_EN	Habilita o uso da região dos dados seguros.
SEC_DATA_AES_EN	Seleciona o algoritmo de encriptação AES para encriptar os dados seguros.
SEC_DATA_INIT_ADDR	Endereço inicial da região dos dados seguros.
SEC_DATA_FINAL_ADDR	Endereço final da região dos dados seguros.
SEC_DATA_KEY_WORD0	Bits 127 a 96 da chave da região dos dados seguros.
SEC_DATA_KEY_WORD1	Bits 95 a 64 da chave da região dos dados seguros.
SEC_DATA_KEY_WORD2	Bits 63 a 32 da chave da região dos dados seguros.
SEC_DATA_KEY_WORD3	Bits 31 a 0 da chave da região dos dados seguros.
SEC_INST_EN	Habilita o uso da região das instruções seguras.
SEC_INST_AES_EN	Seleciona o algoritmo de encriptação AES para encriptar as instruções seguras.
SEC_INST_INIT_ADDR	Endereço inicial da região das instruções seguras.
SEC_INST_FINAL_ADDR	Endereço final da região das instruções seguras.
SEC_INST_KEY_WORD0	Bits 127 a 96 da chave da região das instruções seguras.
SEC_INST_KEY_WORD1	Bits 95 a 64 da chave da região das instruções seguras.
SEC_INST_KEY_WORD2	Bits 63 a 32 da chave da região das instruções seguras.
SEC_INST_KEY_WORD3	Bits 31 a 0 da chave da região das instruções seguras.

Tabela 5.10: Arquivos relacionados com o programa *srec_cipher.c*

Nome do arquivo	Local
srec_cipher.c	*projectrepository/tool_data/srec_cipher/source*
config_file.txt	*projectrepository/tool_data/srec_cipher/config_files*
prog.srec	*projectrepository/tool_data/srec_cipher/stimulus*
cipher_prog.srec	*projectrepository/tool_data/srec_cipher/result*

Tabela 5.11: Número de períodos de clock utilizados na execução de cada programa

Programa	Instruções e dados não seguros	Instruções não seguras e dados seguros	Instruções seguras e dados não seguros	Instruções e dados seguros
prog_01	330	1,28x	1,86x	2,66x
prog_02	15.735	2,11x	1,03x	2,14x
prog_03	806.930	1,00010x	1,00034x	1,00039x
prog_04	1.601.922	1,20019x	1,00020x	1,20054x
prog_05	10.018	1,13x	1,10x	1,23x

Tabela 5.12: Acréscimo de área do LEON3-CONF em relação à área do LEON3

Lógica combinacional	Registradores
57,9%	18,8%

Tabela 5.13: Área da camada de segurança em relação à área do LEON3-CONF

Bloco	Lógica combinacional	Registradores
Camada de segurança (*sec_layer_top*)	36,7%	15,8%

Tabela 5.14: Área de cada bloco da camada de segurança em relação à área total da camada de segurança

Bloco	Lógica combinacional	registradores
sec_layer_top	100,0%	100,0%
| sec_layer_ahbs	6,7%	19,8%
| sec_layer_cd_unit	87,8%	41,9%
| aes_top	84,9%	41,9%
| aes_cipher	31,1%	7,1%
| aes_control	1,1%	13,8%
| aes_decipher	33,8%	7,0%
| aes_inv_key_exp	9,8%	7,0%
| aes_key_exp	9,1%	7,0%
| sec_layer_cdu_sel	2,9%	0,0%
| sec_layer_regs	5,5%	38,3%

Capítulo 6

Conclusões

Em relação as funcionalidades exercitadas pelos programas dos *benchmarks* *benchmark_ asm* (Seção 5.1.3) e *benchmark_ c* (Seção 5.1.4), o LEON3-CONF é equivalente ao LEON3.

Conforme esperado e confirmado pela Tabela 5.11, o número de períodos de clock utilizados na execução de um programa depende do número de instruções (seguras e não seguras), do número de dados (seguros e não seguros) e da maneira como os dados (seguros e não seguros) são acessados pelo programa.

Os programas *prog_ 03* e *prog_ 04* (*benchmark benchmark_ c*), por exemplo, calculam o valor da sequência de Fibonacci F(n) (onde n é 20) utilizando recursão. Nos dois programas, o valor n e o resultado de F(n) são seguros, porém os valores intermediários são seguros apenas no programa *prog_ 04*. Por isso, o tempo de execução dos dois programas com apenas instruções seguras no LEON3-CONF (em relação ao respectivo programa sem instruções e dados seguros) é aproximadamente o mesmo: 1,00034 vezes no caso do *prog_ 03* e 1,00020 vezes no caso do *prog_ 04*. Entretanto, o tempo de execução destes programas com dados seguros no LEON3-CONF (em relação ao respectivo programa sem instruções e dados seguros) é maior no *prog_ 04* do que no *prog_ 03*: 1,20019 vezes no caso do *prog_ 04* e 1,00010 vezes no caso do *prog_ 03* (quando o programa tem instruções não seguras e dados seguros); 1,20054 vezes no caso do *prog_ 04* e 1,00039 vezes no caso do *prog_ 03* (quando o programa tem instruções e dados seguros).

Ainda de acordo com a Tabela 5.11, o tempo de execução de um programa com apenas dados seguros no LEON3-CONF é, no melhor caso, 1,00010 vezes e, no pior caso, 2,11 vezes o tempo de execução do mesmo programa com instruções e dados não seguros; o tempo de execução de um programa com apenas instruções seguras no LEON3-CONF é, no melhor caso, 1,00020 vezes e, no pior caso, 1,86 vezes o tempo de execução do mesmo programa com instruções e dados não seguros; e o tempo de execução de um programa com instruções e dados seguros no LEON3-CONF é, no melhor caso, 1,00039 vezes e, no pior caso, 2,66 vezes o tempo de execução do mesmo programa com instruções e dados não seguros.

Este projeto também comparou a área do LEON3-CONF com a do LEON3 (Tabela 5.12), além de quantificar a área da camada de segurança (Tabela 5.13) e a dos seus blocos (Tabela 5.14). Estas comparações foram realizadas a partir das áreas obtidas pelas sínteses do LEON3 e do LEON3-CONF. Os resultados indicam que a área do LEON3-

CONF é 57,9% maior do que a área do LEON3 em termos da lógica combinacional e 18,8% em termos dos registradores, e que a camada de segurança utiliza 36,7% da lógica combinacional e 15,8% dos registradores do LEON3-CONF. Além disso, verificou-se que a implementação do algoritmo de encriptação e decriptação AES (*aes_top*) utiliza 84,9% da lógica combinacional e 41,9% dos registradores da camada de segurança. Estes últimos dados revelam a importância de novas pesquisas na implementação do algoritmo AES e/ou em novos algoritmos de encriptação e decriptação.

Em resumo, como esperado, se o usuário deseja fornecer confidencialidade à execução e aos dados do seu programa, ele(a) deve estar preparado(a) para aceitar uma perda de desempenho e um custo maior com área que advêm da camanda de segurança da arquitetura do LEON3-CONF.

6.1 Trabalhos Futuros

A arquitetura do LEON3-CONF permite que diferentes algoritmos de encriptação e decriptação sejam facilmente adicionados ao hardware da camada de segurança. Portanto, uma sugestão de trabalhos futuros é adicionar outros algoritmos de encriptação e decriptação ao LEON3-CONF e, então, quantificar e avaliar os seus respectivos desempenhos. Sem dúvida, os scripts e os programas desenvolvidos para esta versão do LEON3-CONF poderão ser utilizados em novas verificações e análises de desempenho.

Outra sugestão é adicionar a garantia de integridade de instruções e/ou dados ao LEON3-CONF.

Referências Bibliográficas

[1] Aeroflex Gaisler AB. *GRMON User's Manual*, novembro 2011. Version 1.1.51.

[2] Cobham Gaisler AB. *Bare-C Cross-Compiler (BCC)*. URL = http://www.gaisler.com/anonftp/bcc/bin/linux/. Versão: BCC 4.4.2 release 1.0.47. Data da instalação: 13 de agosto de 2017. Último acesso: 14 de fevereiro de 2018.

[3] Cobham Gaisler AB. *Download LEON/GRLIB*. URL = http://www.gaisler.com/index.php/downloads/leongrlib. Data do download: 4 de julho de 2011. Último acesso: 16 de dezembro de 2017.

[4] Cobham Gaisler AB. *GRMON2*. URL = http://www.gaisler.com/index.php/products/debug-tools/grmon2. Último acesso: 18 de fevereiro de 2018.

[5] Cobham Gaisler AB. *LEON3 Processor*. URL = http://www.gaisler.com/index.php/products/processors/leon3. Último acesso: 28 de outubro de 2017.

[6] Cobham Gaisler AB. *Bare-C Cross-Compiler (BCC) User's Manual*, dezembro 2017. Version 1.0.50. URL = http://www.gaisler.com/doc/bcc.pdf. Último acesso: 14 de fevereiro de 2018.

[7] Cobham Gaisler AB. *GRLIB IP Core User's Manual*, maio 2017. Version 2017.2. URL = http://www.gaisler.com/products/grlib/grip.pdf. Último acesso: 28 de outubro de 2017.

[8] Cobham Gaisler AB. *GRLIB IP Library User's Manual*, maio 2017. Version 2017.2. URL = http://www.gaisler.com/products/grlib/grlib.pdf. Último acesso: 28 de outubro de 2017.

[9] Cobham Gaisler AB. *GRMON2 User's Manual*, novembro 2017. Version 2.0.87. URL = http://www.gaisler.com/doc/grmon2.pdf. Último acesso: 8 de dezembro 2017.

[10] Cobham Gaisler AB. *LEON/GRLIB Guide - Configuration and Development Guide*, maio 2017. Version 2017.2. URL = http://www.gaisler.com/products/grlib/guide.pdf. Último acesso: 28 de outubro de 2017.

[11] Shaizeen Aga and Satish Narayanasamy. *InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. SIGARCH Comput. Archit. News*, 45(2):94–106, junho 2017.

[12] ARM. *AMBA AXI (Advanced eXtensible Interface) Protocol Specification*, 2008. Version 1.0.

[13] Copyright© 1998-2011 BSI. *GTKWave 3.3 Wave Analyzer User's Guide*, maio 2011. URL = http://www.ic.unicamp.br/~ducatte/mc542/Docs/gtkwave.pdf. Último acesso: 15 de fevereiro de 2018.

[14] Steve Chamberlain and Ian Lance Taylor. *The GNU linker - ld (GNU Binutils)*. Free Software Foundation, Inc., 2013. Version 2.24.0.

[15] Xi Chen, Robert P. Dick, and Alok Choudhary. *Operating System Controlled Processor-Memory Bus Encryption.* In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE'08, pages 1154–1159, New York, NY, USA, 2008. ACM.

[16] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. *Protecting Data on Smartphones and Tablets from Memory Attacks. SIGARCH Comput. Archit. News*, 43(1):177–189, março 2015.

[17] Intel Corporation. *Quartus Prime Pro Edition Help version 17.1 - logic cell Definition.* URL = http://quartushelp.altera.com/17.1/index.htm?textToSearch=#reference/glossary/def_lcell.htm. Último acesso: 29 de abril de 2018.

[18] Intel Corporation. *Quartus Prime Pro Edition Help version 17.1 - Synthesis Resources Reports.* URL = http://quartushelp.altera.com/17.1/index.htm#report/rpt/rpt_file_resource_usage_analysis.htm. Último acesso: 29 de abril de 2018.

[19] Intel Corporation. *Intel® Software Guard Extensions Programming Reference*, outubro 2014. 329298-002US. URL = https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. Último acesso: 26 de maio de 2018.

[20] Guillaume Duc and Ronan Keryell. *CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection.* In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, pages 483–492, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Morris J. Dworkin and et al. *Federal Information Processing Standards Publication (FIPS) 197 - Advanced Encryption Standard (AES). NIST (National Institute of Standards and Technology), U.S. Department of Commerce*, novembro 2001. URL = http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf. Último acesso: 8 de dezembro de 2017.

[22] Intel FPGA. *Intel® Quartus® Prime Software*. URL=https://www.altera.com/products/design-software/fpga-design/quartus-prime/what-s-new.html. Último acesso: 1 de maio de 2018.

[23] Intel FPGA. *Intel® Quartus® Prime Software - Download Center*. URL=https://www.altera.com/downloads/download-center.html. Último acesso: 1 de maio de 2018.

[24] Tristan Gingold. *GHDL*. URL = http://ghdl.free.fr. Versão: ghdl 0.31-1pgavin2-trusty2. Data da instalação: 17 de agosto de 2015. Último acesso: 12 de fevereiro de 2018.

[25] Tristan Gingold. *GHDL Documentation*. URL = https://ghdl.readthedocs.io/en/latest/index.html. Revision 900f7cf3. Último acesso: 13 de fevereiro de 2018.

[26] Barbara Guttman and Edward Roback. *An Introduction to Computer Security: The NIST Handbook (NIST Special Publication 800-12)*. NIST (National Institute of Standards and Technology), U.S. Department of Commerce, Computer Security Division, Computer Systems Laboratory, outubro 1995.

[27] IEEE. *IEEE Internet of Things*. URL=https://iot.ieee.org. Último acesso: 3 de dezembro de 2017.

[28] IEEE. *Towards a definition of the Internet of Things (IoT)*, maio 2015. Revision 1.

[29] Free Software Foundation Inc. *GNU General Public License*. URL=http://www.gnu.org/licenses/gpl.txt. Último acesso: 4 de novembro de 2017.

[30] Free Software Foundation Inc. *GNU Operating System - GNU Binutils*. URL = https://www.gnu.org/software/binutils. Versão: 7 de fevereiro de 2018. Último acesso: 14 de fevereiro de 2018.

[31] SPARC International Inc. *The SPARC Architecture Manual*, 1991. Version 8. Revision SAV080SI9308. URL = http://www.gaisler.com/doc/sparcv8.pdf. Último acesso: 9 de dezembro de 2017.

[32] Terasic Technologies Inc. *Altera DE2-115 Development and Education Board*. URL = http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502&PartNo=1. Último acesso: 4 de novembro de 2017.

[33] Terasic Technologies Inc. *DE2-115 User Manual*, agosto 2017. Version 2.3.

[34] ARM Limited. *AMBA™ Specification*, 1999. Revision 2.0. ARM IHI 0011A. URL = https://developer.arm.com/docs/ihi0011/latest/ambatm-specification-rev-20. Último acesso: 9 de dezembro de 2017.

[35] NIST (National Institute of Standards and Technology), U.S. Department of Commerce. *Final Version of NIST Cloud Computing Definition Published.* URL=https://www.nist.gov/news-events/news/2011/10/final-version-nist-cloud-computing-definition-published. Último acesso: 3 de dezembro de 2017.

[36] NIST (National Institute of Standards and Technology), U.S. Department of Commerce, Computer Security Division, Information Technology Laboratory. *Glossary of Key Information Security Terms*, maio 2013. Editor Richard Kissel; NISTIR 7298, Revision 2. URL = http://nvlpubs.nist.gov/nistpubs/ir/2013/NIST.IR.7298r2.pdf. Último acesso: 14 de outubro de 2017.

[37] NIST (National Institute of Standards and Technology), U.S. Department of Commerce. *Cryptographic Standards and Guidelines - AES Development*, dezembro 2016. URL=https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development. Último acesso: 8 de dezembro de 2017.

[38] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface.* Morgan Kaufmann, 5th edition, 2014.

[39] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. *Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, 2007.

[40] SOURCEFORGE. *GTKWave.* URL = http://gtkwave.sourceforge.net. Versão: gtkwave 3.3.58-1. Data da instalação: 17 de agosto de 2015. Último acesso: 12 de fevereiro de 2018.

[41] William Stallings. *Cryptography and Network Security - Principles and Practice.* Prentice Hall, 5th edition, 2011.

[42] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (GCC) for gcc version 7.3.0. Free Software Foundation, Inc., 2017. *Section 6.32 Specifying Attributes of Variables.*

[43] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. *AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. Proceedings of the 17th Annual International Conference on Supercomputing*, pages 160–171, junho 2003.

[44] Wikipedia (the free encyclopedia). *SREC (file format).* URL = https://en.wikipedia.org/wiki/SREC_(file_format). Versão: 18 de janeiro de 2018. Último acesso: 14 de fevereiro de 2018.

[45] Blaise-Pascal Tine and Sudhakar Yalamanchili. *PageVault: Securing Off-Chip Memory using Page-Based Authentication.* In *Proceedings of the International*

Symposium on Memory Systems, MEMSYS '17, pages 293–304, New York, NY, USA, 2017. ACM.

[46] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. *Improving Cost, Performance, and Security of Memory Encryption and Authentication. SIGARCH Comput. Archit. News*, 34(2):179–190, maio 2006.

[47] Xiaokun Yang and Jean H. Andrian. *A High-Performance On-Chip Bus (MSBUS) Design and Verification. IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1350–1354, julho 2014.

[48] Xiaokun Yang and Wujie Wen. *Design of A Pre-Scheduled Data Bus for Advanced Encryption Standard Encrypted System-on-Chips. Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 506–511, janeiro 2017.

[49] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. *DEUCE: Write-Efficient Encryption for Non-Volatile Memories. SIGPLAN Not.*, 50(4):33–44, março 2015.

Apêndice A

Softwares e Plataforma FPGA

Neste apêndice são documentados os softwares e a plataforma FPGA utilizados neste projeto. O apêndice está estruturado em seis divisões. O Apêndice A.1 especifica o compilador e simulador de VHDL utilizado e os scripts desenvolvidos para executá-lo. O Apêndice A.2 lista o software usado para visualizar as formas de onda geradas pelas simulações dos códigos VHDL. O Apêndice A.3 documenta o *cross-compiler* utilizado para compilar e gerar o executável dos programas em Assembly ou em linguagem C que serão executados no LEON3 ou no LEON3-CONF. O Apêndice A.4 descreve o software usado para sintetizar e programar o LEON3 e o LEON3-CONF na plataforma FPGA e os scripts desenvolvidos para executar este software. O Apêndice A.5 lista as principais características da plataforma FPGA escolhida para este projeto. Por último, o Apêndice A.6 apresenta o monitor de *debug* usado para auxiliar na execução de programas quando o processador (o LEON3 ou o LEON3-CONF) está programado em FPGA.

A.1 GHDL

O compilador e simulador *open-source* para a linguagem VHDL utilizado foi o GHDL[24, 25]. A forma de onda gerada pelo GHDL é do tipo VCD (*Value Change Dump*).

Os scripts *bash* desenvolvidos para executar o GHDL estão descritos na Tabela A.1 e podem ser encontrados no diretório *projectrepository/tool_ data/ghdl/leon3_ sec.*

A.2 GTKWave

O visualizador de forma de onda do tipo VCD (*Value Change Dump*) utilizado foi o GTKWave[40, 13]. O GTKWave é um software livre sob licença GPL (*GNU General Public License*)[29].

A.3 Bare-C Cross-Compiler (BCC)

O *cross-compiler* utilizado para compilar e gerar os executáveis dos programas a serem executados no LEON3 e no LEON3-CONF foi o Bare-C Cross-Compiler (BCC)[6, 2]. O

Tabela A.1: Scripts desenvolvidos para executar o GHDL

Nome do script	Descrição do script
script_00_create_dirs.sh	Este script remove e cria os diretórios utilizados na execução do GHDL.
script_01_ghdl_import.sh	Este script lista e importa todos os arquivos VHDL que compõem o *testbench* e o LEON3-CONF.
script_02_ghdl_make.sh	Este script *analyze* e *elaborate* o *testbench* e o LEON3-CONF.
script_03_ghdl_generate_wave.sh	Este script simula o *testbench* e o LEON3-CONF e grava a forma de onda no diretório *projectrepository/work/waveform/leon3_sec.*

BCC é baseado em *GNU compiler tools*[30]. O diretório de instalação utilizado para o BCC é */opt/sparc-elf-4.4.2*.

As ferramentas do BCC utilizadas estão na Tabela A.2.

Tabela A.2: Ferramentas do BCC

Ferramenta do BCC	Funcionalidade
sparc-elf-gcc -o	Compila e gera o executável do programa em Assembly ou em linguagem C
sparc-elf-objdump	Gera o arquivo *.objdump* a partir do arquivo executável
sparc-elf-objcopy -O srec	Gera o arquivo no formato SREC (Motorola S-record)[44] a partir do arquivo executável

O script em Perl *script_10_srec.pl* foi desenvolvido para executar as ferramentas do BCC (Tabela A.2) de acordo com a linguagem do programa a ser executado no LEON3 ou no LEON3-CONF. O script *projectrepository/progs/bench_01/prog_01/script_10_srec.pl* é um exemplo quando o programa foi desenvolvido em Assembly (*prog.S*), e o script *projectrepository/progs/benchmark_c/prog_01_11/script_10_srec.pl* é um exemplo quando o programa foi desenvolvido em linguagem C (*prog.c*).

No fluxo de desenvolvimento adotado, o formato do programa (em Assembly ou em linguagem C) executado no LEON3 ou LEON3-CONF (quando o processador é simulado ou quando está programado em FPGA) é o formato SREC.

A.4 Intel® Quartus® Prime - Web Edition

O software Intel® Quartus® Prime[22] foi utilizado para sintetizar o LEON3 e o LEON3-CONF e para programá-los em FPGA (Apêndice A.5).

A versão do Quartus usada foi *11.0 Build 157 04/27/2011 SJ Web Edition*. As versões *Web Edition*[23] não requerem licença.

Os scripts da Tabela A.3 foram desenvolvidos para sintetizar o LEON3-CONF e estão no diretório *projectrepository/tool_data/quartus/leon3_sec*.

Tabela A.3: Scripts para sintetizar o LEON3-CONF

Script	Funcionalidade
leon3mp.sdc	Define os clocks, os *false paths*, os *input delays* e os *output delays* para a síntese do LEON3-CONF
script_01.tcl	Contém as definições gerais para a síntese do LEON3-CONF e chama os outros três scripts
script_02.tcl	Lista os arquivos fontes (em VHDL) que compõem o LEON3-CONF
script_03.tcl	Descreve o *assignment* dos pinos da Altera DE2-115

Além disso, os arquivos da Tabela A.4 contêm dicas sobre a utilização do software Intel® Quartus® Prime e estão no diretório *projectrepository/doc/how_to*.

Tabela A.4: Dicas sobre o software Intel® Quartus® Prime

Arquivo	Descrição
how_to_quartus_sof.txt	Arquivo com o passo a passo para sintetizar o LEON3-CONF e, então, gerar o arquivo *leon3mp.sof*
how_to_quartus_prog.txt	Arquivo com o passo a passo para programar o LEON3-CONF (arquivo *leon3mp.sof*) em FPGA (Altera DE2-115)

A.5 Altera DE2-115

A FPGA (*Field Programmable Gate Arrays*) selecionada e utilizada neste projeto foi Altera DE2-115 Development and Education Board[32, 33].

As principais características da placa DE-115 são:

- Altera 60-nm Cyclone IV EP4CE115F29 *device*;
- 115K LEs (*logic elements*);
- 2MB SRAM;
- Duas 64MB SDRAMs;
- 8MB de memória Flash;

- Oscilador de 50MHz.

A.6 *General Debug Monitor* GRMON2

O GRMON2[4, 9] é um monitor de *debug* para os processadores LEON2, LEON3 e LEON4 e para *System-on-Chip* (SoC) *designs* baseados em GRLIB IP library[8].

As principais funções do GRMON2 são:

- Acesso de leitura e de escrita aos registradores e à memória;
- *Built-in disassembler* e *trace buffer management*;
- *Downloading* e execução de aplicações no LEON3;
- Gerenciamento de *breakpoint* e *watchpoint*;
- Conexão remota ao GNU *debugger* (GDB);
- Suporte aos *debug links* USB, JTAG, RS232, PCI, Ethernet e SpaceWire;
- Interface Tcl[1] (scripts, *procedures*, variáveis, *loops* etc.).

O GRMON2 possui dois tipos de licenças: a profissional (*GRMON2 Professional*) e a de avaliação ou acadêmica (*GRMON2 Evaluation/Academic version*). A licença utilizada durante o projeto do LEON3-CONF foi a acadêmica, porque um dos objetivos deste projeto é utilizar ferramentas sob licença GPL ou sob licença acadêmica.

Devido à validade das licenças acadêmicas, algumas licenças do GRMON[1] e algumas do GRMON2 foram utilizadas ao longo deste projeto. A última licença acadêmica do GRMON2 utilizada foi a versão *GRMON2 LEON debug monitor v2.0.85 32-bit eval version*.

No projeto do LEON3-CONF, o GRMON2 foi utilizado para:

- leitura e escrita nos registradores da camada de segurança;
- leitura de determinadas posições da memória (SDRAM);
- leitura do conteúdo da cache de instruções e da cache de dados;
- escrita de um programa no formato SREC[44] na memória (SDRAM) e execução deste programa.

Os comandos do GRMON2 utilizados para cada programa executado no LEON3-CONF em FPGA estão documentados no arquivo *dicas.txt* (Tabela A.5).

[1] *Tool Command Language*

Tabela A.5: Local do arquivo *dicas.txt* para cada *benchmark*

Benchmark	Local do arquivo *dicas.txt*
bench_ 01	*projectrepository/progs/bench_ 01/prog_ */dicas.txt*
benchmark_ regs	*projectrepository/progs/benchmark_ regs/prog_ */dicas.txt*
benchmark_ asm	*projectrepository/progs/benchmark_ asm/prog_ */dicas.txt*
benchmark_ c	*projectrepository/progs/benchmark_ c/prog_ */dicas.txt*

Apêndice B

Exemplo de Configuração do LEON3-CONF

Os programas do *benchmark benchmark_ c* (Seção 5.1.4) contêm a função *sec_ layer_ config* que configura os registradores do LEON3-CONF. Na sequência, a função *sec_ layer_ config* do programa *prog_ 02_ 11/prog.c* é apresentada para exemplificar a configuração do LEON3-CONF.

```
//------------------------------------------------------------------------
// Configure the sec_layer registers
//------------------------------------------------------------------------
void sec_layer_config()
{
  // Security data using AES

  SEC_DATA_CONFIG_REG     = 0x00010001;

  SEC_DATA_INIT_ADDR_REG  = (int)(&vector);
  SEC_DATA_FINAL_ADDR_REG = (int)(&notused_3);

  SEC_DATA_KEY_WORD3_REG  = 0x29F9A0DC;
  SEC_DATA_KEY_WORD2_REG  = 0xAC6E6185;
  SEC_DATA_KEY_WORD1_REG  = 0x5E5020A8;
  SEC_DATA_KEY_WORD0_REG  = 0x7691BE03;

  // Security instruction using AES

  SEC_INST_CONFIG_REG     = 0x00010001;

  SEC_INST_INIT_ADDR_REG  = 0x40000020;
  SEC_INST_FINAL_ADDR_REG = 0x400001AC;

  SEC_INST_KEY_WORD3_REG  = 0x09CF4F3C;
```

```
  SEC_INST_KEY_WORD2_REG  = 0xABF71588;
  SEC_INST_KEY_WORD1_REG  = 0x28AED2A6;
  SEC_INST_KEY_WORD0_REG  = 0x2B7E1516;

  // Performance configuration

  SEC_PERF_INIT_ADDR_REG  = 0x40000020;
  SEC_PERF_FINAL_ADDR_REG = 0x400001B0;

  // Enable the performance analysis
  SEC_PERF_CONFIG_REG     = 0x01;

  // Enable security module
  SEC_CONFIG_REG          = 0x01;
}
```