PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
ENGENHARIA DE COMPUTAÇÃO

PESQUISA E DESENVOLVIMENTO DE MECANISMOS DE COMUNICAÇÃO SEGURA ENTRE PROCESSADORES UTILIZANDO TECNOLOGIA TRUSTZONE

HÉLIO SOUZA FUQUES FILHO

Monografia apresentada como requisito parcial à obtenção do grau de Engenheiro de Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2019

PESQUISA E DESENVOLVIMENTO DE MECANISMOS DE COMUNICAÇÃO SEGURA ENTRE PROCESSADORES UTILIZANDO TECNOLOGIA TRUSTZONE

RESUMO

Com o aumento da utilização de processadores embarcados para realizar operações de alto valor agregado há uma crescente necessidade de investimentos em segurança. Os processadores ARM utilizam uma extensão de segurança chamada TrustZone, a qual permite implementar um ambiente de execução confiável (*Trusted Execution Enviroment*). Nesse contexto, esse trabalho compreende o estudo do processador ARM, a tecnologia TrustZone e a implementação de um sistema de comunicação que utiliza a tecnologia TrustZone. A implementação do sistema foi realizada utilizando a ferramenta *Open Virtual Platform* e transfere informações para um processador com a extensão de segurança habilitada, o qual armazena os dados enviados a ele em um espaço de memória protegido, utilizando a tecnologia TrustZone.

Palavras-Chave: ARM, TrustZone, Segurança.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AMBA – *Advanced Microcontroller Bus Architecture*

API – *Application Programming Interface*

APB – *Advanced Peripheral Bus*

AXI – *Advanced eXtensible Interface*

CPU – *Central Processing Unit*

GIC – *Generic Interrupt Controller*

IP – *Intellectual Property*

OS – *Operating System*

OVP – *Open Virtual Platafoms*

RAM – *Random-Access Memory*

RISC – *Reduced Instruction Set Computer*

SOC – *System on Chip*

TZASC – *TrustZone Address Space Controller*

TZMA – *TrustZone Memory Address*

TZPC – *TrustZone Protection Controlle*

DMA – *Direct Memory Access*

1. INTRODUÇÃO

Os equipamentos embarcados realizam cada vez mais operações de muito valor agregado, tais como operações bancárias ou aplicações críticas de tempo real. Esses equipamentos executam software que permitem aos usuários baixarem e utilizarem aplicações de terceiros, e além disso, há o aumento do valor agregado dos dados trafegados através destes equipamentos. Estas características motivam *hackers* a invadir estes sistemas, visando o roubo de informações sensíveis ou a interrupção da operação dos mesmos buscando benefícios financeiros (e.g. *ransomware*) [ARM09]. Com o aumento dos software embarcados para diversas aplicações voltadas para a internet das coisas (IoT), esses aparelhos precisam de soluções para os protegerem das mais diversas vulnerabilidades.

Estes sistemas usam processadores embarcados, memórias, barramentos e diversos periféricos, gerando muitos pontos de vulnerabilidade. Por consequência, segurança deve ser um requisito de projeto. Os processadores ARM utilizam uma extensão de segurança no nível de hardware e software denominada *ARM TrustZone*. Essa tecnologia permite que um sistema mantenha dados sensíveis seguros armazenados, mesmo que o processador execute aplicações não seguras [ARM09].

Segurança na computação é a proteção dos hardwares e software, objetivando a redução de vulnerabilidades nesses dispositivos [PP15]. Em segurança de computadores, uma vulnerabilidade é uma fraqueza que permite que um atacante reduza a garantia da informação de um sistema. Vulnerabilidade é a interseção de três elementos: uma suscetibilidade ou falha do sistema, acesso do atacante à falha e a capacidade do atacante de explorar a falha. Para explorar uma vulnerabilidade, um atacante deve ter pelo menos uma ferramenta ou técnica aplicável que possa conectar-se a uma fraqueza do sistema. Desta forma, vulnerabilidade também é conhecida como superfície de ataque.

Agregar em um produto embarcado técnicas de segurança aos ataques externos implica em custo de projeto, tanto de hardware quanto o software [ARM09]. O livro *Computer Security* [PP15] descreve 3 propriedades básicas de segurança que devem ser consideradas como pilares para um sistema seguro:

- **Integridade** - prevenção de que uma informação não será modificada ou destruída por uma entidade não autorizada;
- **Disponibilidade** - capacidade do sistema de permitir que um dispositivo seja utilizado apenas por entidades autorizadas;
- **Confidenciabilidade** - propriedade de não exibir informações para entidades não autorizadas e identificar possíveis falhas de segurança.

Além dessas propriedades básicas, o livro traz também duas propriedades adicionais que devem ser consideradas principalmente em redes de comunicação:

- **Autenticação** - Capacidade do sistema de garantir que o originador de uma requisição a um componente seguro possua as credenciais que comprovem a sua identidade;
- **Não-repúdio** - Conceito que refere-se a um serviço que fornece prova da origem dos dados e da integridade dos mesmos. Para isto, é necessário a comunicação com entidades autenticadoras (por exemplo, um servidor Radius).

Um sistema normalmente possui a sua segurança implementada somente no nível de software. Quando um sistema é projetado considerando as propriedades de segurança, tanto no nível de software quanto no nível de hardware, um alto nível de segurança é alcançado [NMB+16]. Este conceito é denominado de *Trusted Execution Environments* (TEE). A Global Platform Compliance Program (https://globalplatform.org) define TEE como uma área segura no principal processador de um dispositivo e que ele deve oferecer a execução de software autorizados [NMB+16].

O artigo "Building a Secure System using TrustZone Technology" [ARM09] apresenta um exemplo de pagamento móvel para ilustrar como esses conceitos de segurança são implementados por um SoC utilizando TrustZone. O exemplo de uma máquina de cartão de crédito móvel traz a necessidade de um sistema que permita o armazenamento de um grande volume de dados, e que estes somente possam ser acessados após um processo de autenticação. É importante ressaltar que o ponto de maior vulnerabilidade do sistema é quando os dados sigilosos são manipulados. A extensão de segurança permite instalar o software de comunicação com o cartão, o teclado e o *display* que irá receber os dados em memória segura garantindo a integridade e confidenciabilidade na interação com esses dispositivos. A maior falha de segurança está na comunicação com o cartão que, apesar de poder ocorrer no mundo seguro, irá precisar de alguma camada de criptografia na comunicação.

1.1 Motivação

A principal motivação deste trabalho é aplicar os conhecimentos adquiridos ao longo do curso de Engenharia de Computação na área de arquitetura de computadores, microcontroladores e sistemas operacionais para a solução de possíveis vulnerabilidades na comunicação de processadores ARM, investigando meios de implementar conceitos de segurança nesses processadores utilizando uma extensão nativa do fabricante. Uma lacuna observada no curso é a parte relativa aos métodos de segurança em processadores, e em particular em como agregar estes métodos em sistemas embarcados.

1.2 Objetivos

Esse trabalho descreve a implementação de uma plataforma simulada utilizando a ferramenta *Open Virtual Processors* para realizar a comunicação entre processadores ARM Cortex-A8 e de um periférico para gerenciar essa comunicação. A implementação utiliza TrustZone para criar um ambiente seguro em um dos processadores da plataforma e validar a proteção gerada por esse mecanismo.

O objetivo estratégico deste TCC é estudar e compreender a tecnologia ARM TrustZone e a sua utilização para realizar a comunicação entre processadores.

Dentre os objetivos específicos elenca-se:

- estudar e compreender o processador ARM e sua extensão de segurança (TrustZone);
- compreender o ambiente de modelagem de plataformas de hardware, Open Virual Platform (OVP) [IMP19b];
- realizar uma prova de conceito de um ambiente seguro entre processadores independentes para que eles possam trocar informações sigilosas utilizando os componentes de segurança dos processadores ARM com a extensão TrustZone.

1.3 Organização do documento

O volume final deste TCC está organizada em 5 capítulos:

- **Capítulo 1** - descreve o tema, motivação e objetivos deste trabalho de conclusão.
- **Capítulo 2** - apresenta o estudo da ferramenta que o trabalho de conclusão utiliza.
- **Capítulo 3** - apresenta a arquitetura do processador ARM, os aspectos de hardware e software da extensão de segurança TrustZone e descreve o processador Cortex-A8.
- **Capítulo 4** - descreve o desenvolvimento da plataforma prova-de-conceito desse TCC para um sistema de comunicação utilizando o TrustZone.
- **Capítulo 5** - apresenta as conclusões deste TCC.

2. OVP - OPEN VIRTUAL PLATAFORMS

Open Virtual Plataforms (OVP), da empresa Imperas, é um conjunto de ferramentas para modelar plataformas de hardware de forma virtual, visando acelerar o desenvolvimento de software embarcado. O OVP possui três componentes principais: (*i*) uma API que permite a modelagem em C de processadores e periféricos; (*ii*) uma biblioteca *opensource* com diversos processadores e periféricos; (*iii*) simulador OVPsim, que permite a emulação e execução dos modelos de processadores e periféricos.

O OVP disponibiliza APIs específicas para a modelagem de processadores e periféricos. Para a modelagem de processadores a API *VMI* é utilizada e, para a criação de periféricos, é utilizada as APIs *PPM* e *BHM*.

A biblioteca *open-source*, também chamada de *Open Source Models*, fornece objetos pré-compilados e arquivos fontes de modelos de processadores como ARM, MIPS, Power PC, entre outros, assim como diferentes tipos de componentes como RAM, ROM, cache e *bridges*. Esses modelos são disponibilizados em plataformas pré-projetadas, as quais permitem a execução de diferentes sistemas operacionais [IMP19b].

O OVPSim é a ferramenta que permite a simulação dos processadores e periféricos de forma rápida e eficiente [IMP19b]. O OVPsim realiza a transformação binária de forma dinâmica das instruções da arquitetura alvo para as instruções do computador hospedeiro.

Junto ao OVPSim é disponibilizado o Imperas *IGen model building wizard*, ferramenta utilizada para o desenvolvimento deste trabalho. Esse software recebe um *script* escrito na linguagem TCL e cria uma plataforma virtual com os modelos para serem executados no OVPSim. Sendo assim, utilizando uma linguagem de *script* de alto nível, o IGen cria o código C OVP com as funções das APIs, gerando as interfaces para os periféricos e os modelos de funções necessárias para receber entradas do usuário (por exemplo, a "casca" da função relativa a um periférico, já integrada ao processador).

2.1 Criação de uma plataforma com processador ARM

Esta seção descreve o processo para gerar uma plataforma com um processador ARM utilizando o IGen. O exemplo dessa seção é baseado nos exemplos disponibilizados pela Imperas [Imp19a].

Para a simulação de uma plataforma embarcada utilizando o OVP, dois arquivos devem ser criados: o modelo de hardware, e o software que será executado sobre a plataforma, denominado como aplicação.

O código apresentado na listagem 2.1 é utilizado para gerar o modelo com o processador ARM e uma memória RAM. O primeiro comando TCL atribui um nome para a plataforma – *armExample*. Logo após, é criado um barramento com largura de 32 bits. Posteriormente, o comando *ihwaddprocessor* instancia o processador, utilizando os seguintes parâmetros:

- *instancename*: nome identificador do processador;
- *vendor*: fornecedor do processador;
- *library*: nome da biblioteca à qual o processador pertence;
- *type*: nome do processador;
- *version*: versão do processador, no caso, Cortex-A9UP;
- *variant*: versão do processador;
- *semihostname*: nome da biblioteca;
- *semihostvendor*: fornecedor da biblioteca.

Na sequência, utilizando o comando *ihwconnect*, o processador é conectado ao barramento de 32 bits criado anteriormente nas portas de instrução e de dados. Finalmente, é criado a memória RAM que então é conectada ao processador.

Código 2.1: TCL utilizado pelo iGen para gerar uma plataforma com um processador ARM.

```
1  ihwnew -name armExample
2  ihwaddbus -instancename mainBus -addresswidth 32
3  ihwaddprocessor -instancename cpu1
4                  -vendor arm.ovpworld.org
5                  -library processor
6                  -type arm -version 1.0
7                  -variant Cortex-A9UP
8                  -semihostname armNewlib
9                  -semihostvendor arm.ovpworld.org
10 ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
11 ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA
12 ihwaddmemory -instancename ram1 -type ram
13 ihwconnect -bus mainBus -instancename ram1 -busslaveport sp1 -loaddress 0x0
14     -hiaddress 0xffffffff
```

Para gerar a plataforma a ser executada no OVPSim, inicialmente é necessário compilar a aplicação utilizando o compilador para o processador definido no modelo de hardware (denominado *módulo*). O módulo é compilado utilizando o arquivo makefile do OVP. Para exemplificar a criação e execução de um software embarcado nesta plataforma, um simples código *Hello World* é utilizado como aplicação. Para executar a simulação é utilizado o *harness.exe*, conforme a linha abaixo, definindo o arquivo de módulo e a aplicação a ser executada.

```
harness.exe --modulefile module/model.so
        --program application/application.ARM_CORTE_A.elf
```

A Figura 2.1 apresenta o diagrama de blocos referente ao hardware simulado pelo código 2.1.

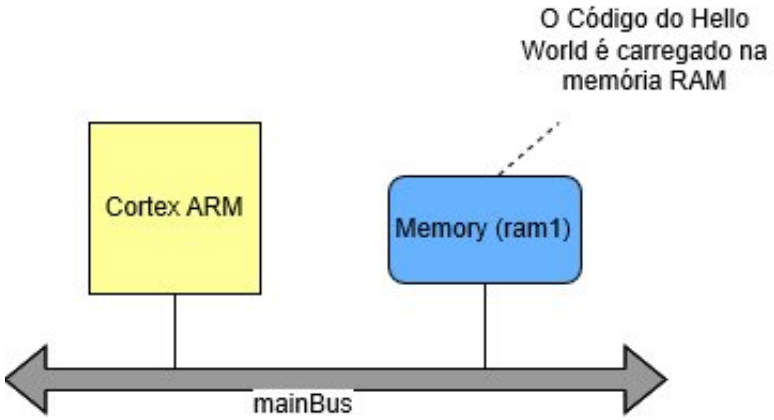

Figura 2.1: Plataforma de Hardware simples gerada com um processador ARM e uma memória RAM.

2.2 Execução de um software utilizando a extensão de segurança TrustZone

Para demonstrar como criar uma plataforma utilizando um processador ARM com a extensão TrustZone, utiliza-se um exemplo fornecido pela OVP [Imp18].

Para uma plataforma com o processador ARM utilizar a extensão de segurança TrustZone, basta controlar um bit que indica para cada barramento de acesso se o mesmo é seguro ou não [Imp18]. No barreamento AXI, esse bit é o sinal AxPROT[1]. Para acessos seguros, o sinal AxPROT[1] deve ser 0 e para acessos não seguros deve ser 1 [Imp18]. Para uma plataforma que utiliza o TrustZone, um barramento ponte é utilizado para criar uma mapa entre os endereços seguros e os não seguros [Imp18].

Nos modelos do processador ARM que possuem a extensão TrustZone, o bit 40 de um endereço é o bit NS (Non-Secure) correspondente ao sinal AxPROT[1]. Utiliza-se o

bit 40 porque os bits 31:0 são os de endereçamento padrão para 32 bits, e os bits 39:23 são usados quando o processador suporta o *Large Physical Address Extension* (LPAE). Para simplificar as implementações, o bit 40 é sempre utilizado, independente se LPAE está presente ou não na plataforma.

O código 2.2 foi retirado do documento que descreve a modelagem com TrustZone da Imperas [Imp18]. Esse código descreve a criação do modelo que irá habilitar o modo seguro na sua aplicação.

Na linha 7, do código 2.2 da plataforma segura, um barramento nomeado *tzBus* com 41 bits é criado. Na linha 12 é criada o barramento principal, *pBus*, que conecta o processador aos dispositivos de E/S e à memória. Uma ponte (*bridge*) na linha 17 é conectada aos dois barramentos. A *bridge* é utilizada para mapear os endereços seguros do *tzBus* para o barramento principal *pBus*, fazendo com que qualquer endereço possa ser acessado pelo modo seguro [Imp18]. Esta *bridge* é criada para que o bit 40 possa ser alterado.

Nas linha 24 e 25, uma memória RAM é definida e conectada ao barramento principal *pBus*. Nas linhas subsequentes até a 32, uma ponte é criada com o *tzBus* para definir esse endereço como sempre seguro. O restante do código basicamente instancia o processador e o o periférico UART nos endereços seguros.

Código 2.2: TCL para uma plataforma contendo uma memória RAM e um UART.

```
1  ihwnew -name alwaysSecure
2  iadddocumentation -name Description -text $desc
3
4  #
5  # tzBus - 41 bits of address - implements secure and non-secure address spaces
6  #
7  ihwaddbus -instancename tzBus -addresswidth 41
8
9  #
10 # pBus  - 32 bits of address - physical bus connected to devices and memory
11 #
12 ihwaddbus -instancename pBus  -addresswidth 32
13
14 #
15 # Bridge - Secure space of tzBus always mapped to pBus
16 #
17 ihwaddbridge -instancename secure
18 ihwconnect   -instancename secure -busslaveport  sp -bus tzBus -loaddress 0x00000000
   ↪  -hiaddress 0xffffffff
```

```
19 ihwconnect   -instancename secure -busmasterport mp -bus pBus -loaddress 0x00000000
   ↪  -hiaddress 0xffffffff
20
21 #
22 # Memory: 0x00000000    0x001fffff  RAM
23 #
24 ihwaddmemory -instancename ram0 -type ram
25 ihwconnect   -instancename ram0 -busslaveport sp1 -bus pBus -loaddress 0x00000000
   ↪  -hiaddress 0x001fffff
26
27 #
28 # Bridge - Memory always accessible from non-secure mode
29 #
30 ihwaddbridge -instancename ram0NS
31 ihwconnect   -instancename ram0NS -busslaveport  sp -bus tzBus -loaddress
   ↪  0x10000000000 -hiaddress 0x100001fffff
32 ihwconnect   -instancename ram0NS -busmasterport mp -bus pBus -loaddress 0x00000000
   ↪  -hiaddress 0x001fffff
33
34 #
35 # Processor
36 #
37 ihwaddprocessor -instancename cpu -type arm -vendor arm.ovpworld.org -endian little
   ↪  -semihostname armNewlib -semihostvendor arm.ovpworld.org
38 ihwconnect      -instancename cpu -busmasterport INSTRUCTION -bus tzBus
39 ihwconnect      -instancename cpu -busmasterport DATA        -bus tzBus
40 ihwsetparameter   -handle       cpu -name variant       -value Cortex-A9UP   -type
   ↪  enum
41 ihwsetparameter   -handle       cpu -name compatibility -value nopSVC        -type
   ↪  enum
42
43 #
44 # UART:   0x10000000    0x10000fff  UartPL011
45 #
46 ihwaddperipheral -instancename uart0   -type UartPL011 -vendor arm.ovpworld.org
47 ihwconnect       -instancename uart0   -busslaveport  bport1 -bus pBus -loaddress
   ↪  0x10000000 -hiaddress 0x10000fff
48 ihwsetparameter    -handle       uart0   -name outfile -value uart0.log  -type string
49
50 #
51 # Add command line arguments to platform
```

```
52 #
53 ihwaddclp -allargs
```

A Figura 2.2 mostra que ao final da descrição em TCL, a plataforma terá no barramento *tzBus* a faixa de endereços 0x0000000 a 0x001FFFFF para a memória segura, 0x1000000 a 0x10000FFF como faixa de endereços seguro para escrita na UART, e o endereçamento não seguro de 0x1000000000 até 0x100001FFFFF com uma ponte para a mesma memória conectada na faixa de endereço da memória segura.

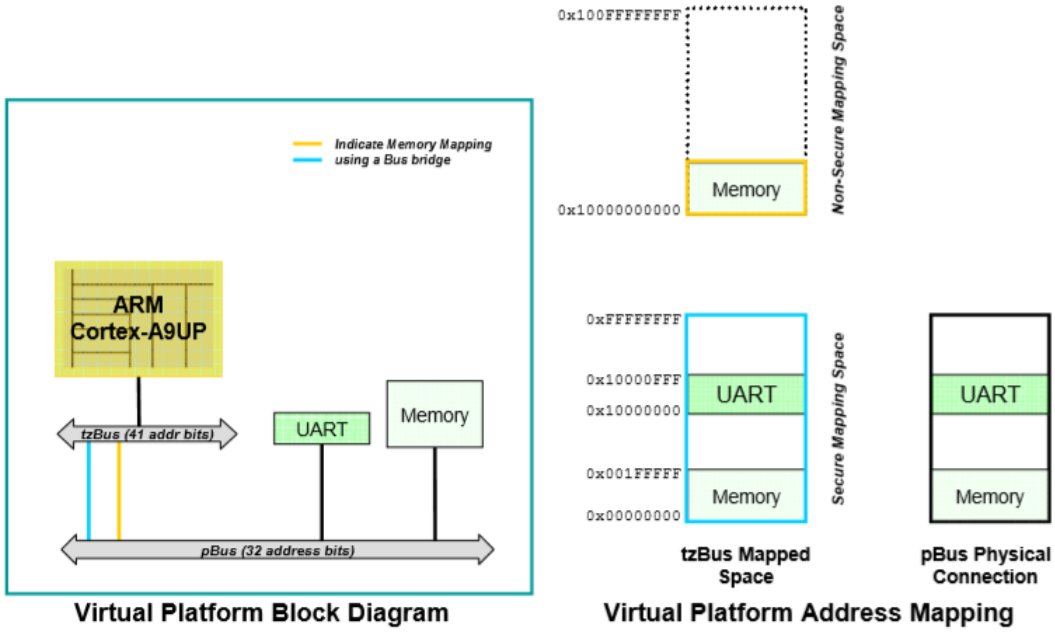

Figura 2.2: Plataforma de Hardware gerada com um processador ARM e uma memória RAM [ARM09].

O código 2.3 é utilizado nesse exemplo para escrever na UART, a qual está mapeada na região segura de memória. As linhas 4 e 5 configuram o mapeamento da memória virtual para a memória física e a configuração do *translation lookaside buffer* (TLB). Logo após, o método *putString* escreve na UART. Nas linhas 10 e 13, o bit NS é ativado e a TLB atualizada, fazendo com que a linha 16 seja executada em modo não seguro, emitindo assim, uma exceção no processador, já que a UART está configurada em um espaço de endereçamento no mundo seguro.

Código 2.3: Código de interação com a UART.

```
1  int main() {
2      // Setup and enable the secure mode TLB
3      // Note: Processor starts out in secure mode
4      setupTranslationTable();
5      enableTLB();
```

```
6       printf ("\n*** Writing to uart from Secure Mode\n");
7       putString("Hello from secure mode!\n");
8
9       printf ("\n*** Entering Non-Secure Mode\n");
10      enterNonSecure();
11
12      // Enable the non-secure TLB (uses same translation table as secure mode)
13      enableTLB();
14
15      printf ("\n*** Writing to uart from Non-Secure Mode\n");
16      putString("Hello from non-secure mode!\n");
17  }
```

A figura 2.3 mostra a saída do terminal após a simulação do plataforma executando o código 2.3. Como pode ser visto na imagem, a execução das linhas 4 até 9 do código ocorrem sem problemas. Quando o processador ativa o endereçamento para memórias não seguras e altera o bit NS para 1 na linha 10, o processador emite uma erro (*Processor Exception*), por não possuir acesso aos endereços em que a UART se encontra.

```
platform/platform.Linux64.exe \
      --program application/application.ARM_CORTEX_A9.elf \
      --excludem ARM_MORPH_BII \
    | tee imperas.log


CpuManagerMulti (64-Bit) v20170201.0 Open Virtual Platform simulator from www.IMPERAS.com.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

CpuManagerMulti started: Mon Sep 16 19:44:43 2019



*** Writing to uart from Secure Mode

*** Entering Non-Secure Mode

*** Writing to uart from Non-Secure Mode
Processor Exception (PC_PRX) Processor 'alwaysSecure/cpu' 0x823c: e5d33000 ldrb    r3,[r3]
Processor Exception (PC_RPX) No read access at 0x10000018
Processor Exception (PC_SED) NOTE: simulated exceptions are not enabled on processor alwaysSecure/cpu. If an application is being sim
ulated that requires simulated exception support (to map memory on demand using an MMU, for example) please ensure simulated exceptio
ns are enabled for correct behavior.

CpuManagerMulti finished: Mon Sep 16 19:44:43 2019


CpuManagerMulti (64-Bit) v20170201.0 Open Virtual Platform simulator from www.IMPERAS.com.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.


*** uart0.log: ***
Hello from secure mode!
```

Figura 2.3: Execução do exemplo de código de escrita na UART.

Como pode ser visto, o OVP permite a simulação e criação de um ambiente seguro. Sendo assim, este trabalho utiliza o OVP para modelar o ambiente das plataformas para se comunicar em modo seguro.

2.3 Modelagem de um Periférico

Nessa Seção é exemplificado como criar um periférico para ser utilizado em uma plataforma no OVP, mostrando um exemplo baseado no periférico *DynamicBridge* [Imp18].

Para a criação de um periférico para o OVP, um arquivo em TCL deve descrever o periférico. A partir desse arquivo, o iGen gera o código para implementar o hardware do periférico descrito e um arquivo para implementar o comportamento do periférico(software do periférico).

O código apresentado na listagem 2.4 gera o periférico. O primeiro comando atribui o nome do periférico e as suas informações. Logo após, são criados dois barramentos, *slave* e *master*, ambos com o parâmetro *-mustbeconnected* que irá obrigar a conexão desses barramentos quando esse periférico for instanciado na plataforma.

Na linha 13 é criada a porta *enable*, com a chamada de função de *callback* definida pelo parâmetro $-$*updatefunction*. Essa função será gerada pelo iGen e então preenchida pelo projetista para gerar o comportamento da porta. Ao final do código, são definidos parâmetros que serão passados ao periférico no momento da sua instancia na plataforma.

Código 2.4: TCL do Dynamic Bridge.

```
1  imodelnewperipheral -imagefile pse.pse -name  DynamicBridge  -library peripheral -constructor co
2
3  set outputBusWidth  32
4
5  # Define bus slave input port
6  imodeladdbusslaveport  -name sp -remappable -mustbeconnected
7
8  # Define bus master output port
9  imodeladdbusmasterport -name mp -addresswidth $outputBusWidth -mustbeconnected
10
11 # Define enable net input port
12 imodeladdnetport -name enable -type input -updatefunction updateEnable
13
14 # Define formals for specifying the output master and slave bus port location and size
15 imodeladdformal -name mpLoAddress -type integer
16 imodeladdformal -name spLoAddress -type integer
17 imodeladdformal -name portSize    -type integer
```

O iGen gera os arquivos que implementam as estruturas do código TCL acima, e o arquivo que deve ser complementado para implementar as funcionalidades do periférico. O código 2.5 mostra a implementação da função que irá ser chamada quando alguma alteração for feita na porta *enable*. A linha 15 cria uma ponte entre os barramentos do periférico, e a linha 9 deleta essa ponte. Estes eventos ocorrem de acordo com o valor escrito na porta.

Código 2.5: TCL do Dynamic Bridge.

```
1  ...
2  PPM_NET_CB(updateEnable) {
3      Bool enable  =  value && !bridgeEnabled;
4      Bool disable = !value &&  bridgeEnabled;
5
6      if (disable) {
7
8          // Disable dynamic bridge
9          ppmDeleteDynamicBridge("sp", spLoAddress, portSize);
10         bridgeEnabled = False;
11
12     }
13     if (enable) {
14        // Enable dynamic bridge
15        ppmCreateDynamicBridge("sp", spLoAddress, portSize, "mp", mpLoAddress);
16        bridgeEnabled = True;
17
18     }
19 ...
```

3. ARQUITETURA ARM COM SUPORTE A TRUSTZONE

A sigla ARM refere-se a "*Advanced RISC Machine*"e, como o próprio nome da família de processadores se refere, eles são processadores de arquitetura RISC (Reduced Instruction Set Computer). Os processadores RISC partem do princípio que as instruções devem ser extremamente simples [Bak16], que utilizem operações *load-store* para acesso à memória, e as operações lógicas e aritméticas sejam realizadas entre registradores.

As instruções da arquitetura ARM, assim como instruções RISC de modo geral, são divididas em três principais tipos de instruções: aritméticas, de memória e de controle. Os programadores de um processador ARM devem explicitamente carregar e selecionar os registradores que serão utilizados em cada instrução, incluindo também os registradores utilizado em resultados intermediários. O programador tem total controle do estado dos registradores [Bak16].

Desde 1983, oito versões da arquitetura ARM foram criadas. Elas são nomeados sempre colocando um sufixo com a versão (ARMv1, ARMv2, ARMv3, etc.) [NMB+16]. Cada versão da arquitetura possui diferentes perfis. A arquitetura ARMv7, por exemplo, provê três perfis [Bak16]:

- ARMv7-A: perfil de propósito geral;
- ARMv7-R: perfil utilizado para aplicações de tempo real;
- ARMv7-M: perfil otimizado para microcontroladores.

A arquitetura ARM trabalha com três endereços distintos, pois uma instrução pode referenciar até 3 registradores simultaneamente. O programador possui acesso a 16 registadores de propósito geral, r0 até r15. Abaixo segue a definição destes [Bak16]:

- r0 e r3: mantém valores de argumentos passados para uma sub-rotina e também para manter resultados retornados de uma sub-rotina;
- r4 até r11: são registradores de propósito geral;
- r12: registrador temporário utilizado na chamada de funções;
- r13: registrador de topo da pilha (SP);
- r14: registrador de retorno de função (*link*);
- r15: contador do programa (PC).

3.1 TrustZone

TrustZone é uma extensão de segurança de hardware que provê a todo o sistema um particionamento seguro para as aplicações sendo executados. A extensão de segurança TrustZone utiliza o conceito de particionar o sistema em "dois mundos diferentes", o *secure world* e o *normal world*. O processador pode executar funções em dois modos: o modo seguro e o modo não seguro. Tudo que é executado quando o processador está em modo seguro é chamado de *secure world*, e tudo que é executado no modo não seguro é chamado de *normal world*. O mundo seguro permite a construção de um ambiente isolado para a execução de aplicações seguras [NMB+16].

3.1.1 Aspectos de Hardware do TrustZone

Para garantir essa divisão de "mundos", o processador ARM possui diversos IPs (módulos de propriedade intelectual) voltados para segurança. Abaixo seguem os periféricos que suportam as implementações de segurança [NMB+16]:

- Advanced eXtensble Interface (AXI) bus - barramento principal do processador ARM. O barramento possui um bit que indica se uma leitura ou escrita será feita em uma memória segura ou não segura;
- Advanced Peripheral Bus (APB) - barramento de baixo desempenho utilizado para interface aos periféricos mais lentos;
- AXI to APB Bridge - ponte que permite a comunicação entre a CPU e os seus periféricos de forma segura. A *bridge* verifica as permissões e bloqueia o acesso caso haja acessos não autorizados;
- Cache Controller - a cache utiliza um bit adicional (33º bit) para indicar se um endereço de memória é de uma memória segura ou não;
- Direct Memory Access (DMA) Controller - responsável por transferir dados para memórias físicas. Esse componente pode realizar transferências tanto em modo seguro e não seguro. Ele também previne acesso não seguro a memória segura.

A lista abaixo apresenta os componentes que implementam as medidas de segurança do TrustZone [NMB+16].

- TrustZone Address Space Controller (TZASC) - controlador que realiza a classificação das memórias escravas mapeadas pelo barramento AXI como seguro ou não seguro. O TZASC permite a distinção de unidades seguras e não seguras.

- TrustZone Memory Address (TZMA) - Realiza a divisão da memória em partições múltiplas de 4 kB e não supeior a 2MB.
- Generic Interrupt Controller (GIC) - periférico responsável por gerenciar as interrupções seguras e não seguras. Responsável também por não permitir interrupções provenientes de acessos não autorizados.
- TrustZone Protection Controller (TZPC) - Periférico de controle que possui três registradores de 2 bits para controlar até 8 periféricos que poderão transitar entre o mundos seguro e o não seguro.

A Figura 3.1 ilustra a ligação do TZPC com os demais componentes de hardware do processador ARM. O AXI-to-APB Bridge na figura abaixo está controlando 4 periféricos. O TZPC é configurado como seguro, o RTC como não seguro e as interfaces de mouse e teclado (KMI) possuem a sua segurança trocada de acordo com a execução do software. O TZPC poderá em tempo de execução do software mudar os sinais de entrada do AXI-to-APB bridge para transitar o KMI entre os modo seguro e não seguro. O sinal DECPROT representa o bit de segurança. Como pode ser visto na figura, o TZPC pode dinamicamente alternar o modo de operação do periférico KMI.

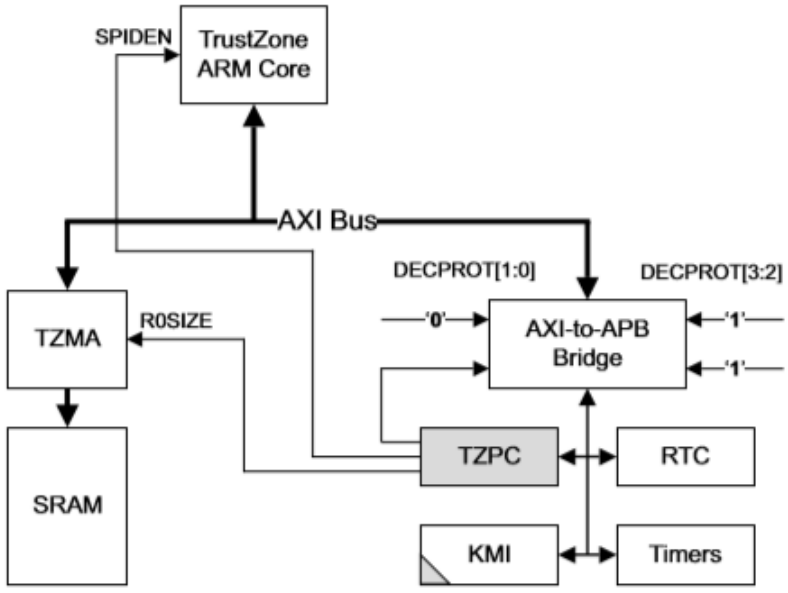

Figura 3.1: Interação do TZPC para controlar sinais de segurança em um SoC [ARM09].

3.1.2 Aspectos de Software do TrustZone

Para utilizar as capacidades providas pelos IPs de segurança, é necessário realizar a implementação em software para ser executada em modo seguro. Essas implementações

podem ser feitas utilizando uma arquitetura baseada em um sistema operacional seguro e dedicado, uma biblioteca que realiza as chamadas pontuais seguras ou simplesmente algumas funções disponíveis no firmware do sistema operacional [NMB+16].

O artigo *ARM Security Technology - Building a Secure System using TrustZone Technology* [ARM09] detalha como implementar um software em processadores ARM que utiliza os recursos TrustZone usando tanto uma simples biblioteca com chamadas para o "mundo seguro", quanto através de um sistema operacional.

Utilizar um sistema operacional (SO) como solução para gerenciar o software no mundo seguro é uma solução complexa e robusta para o sistema, pois permite a execução de múltiplas aplicações seguras simultaneamente [ARM09]. As arquiteturas que utilizam um SO normalmente fazem com que cada software execute em um processador virtual com um SO independente, e que cada mundo use as interrupções de hardware para poder utilizar o processador. Na Figura 3.2 pode ser visto uma possível arquitetura para uma aplicação que utiliza um SO seguro como solução para realizar a implementação de software no mundo seguro.

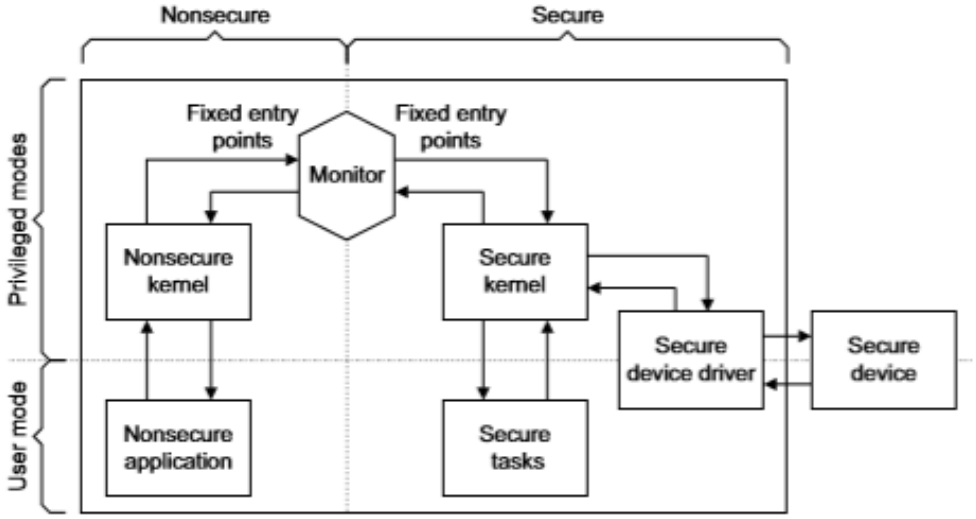

Figura 3.2: Sistemas operacionais seguros [ARM10].

Em contrapartida ao uso de um SO no mundo seguro, a implementação de software para utilizar o TrustZone pode ser feita utilizando uma biblioteca que gerencia todas as chamadas tanto para o mundo seguro quanto para o mundo normal.

3.2 Cortex-A8

A implementação em ambiente simulado utiliza a modelagem de um processador ARM Cortex-A8. O Cortex-A8 implementa a arquitetura ARMv7-A 32bits. A figura abaixo ilustra o diagrama de blocos do processador.

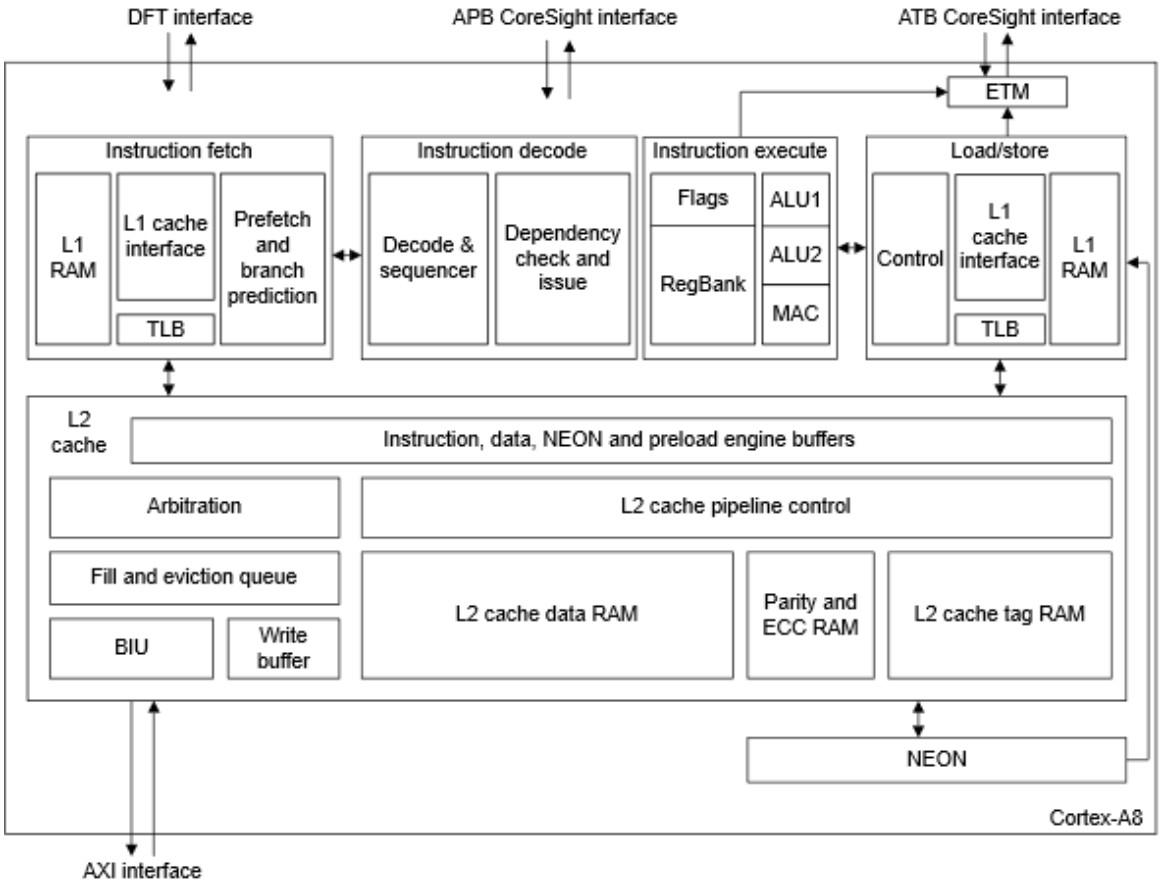

Figura 3.3: Diagrama de blocos do processador Cortex-A8 [ARM10].

Os processadores da arquitetura ARMv7 possuem um registrador para realizar configurações e leituras de controle do mesmo – CPSR (*Current Program Status Register*). Dentre essas, configurações podem ser destacadas as informação de operações lógicas/aritméticas, interrupções e configuração dos modos de operação do processador. A Figura 3.4 ilustra os bits de configuração desse registrador.

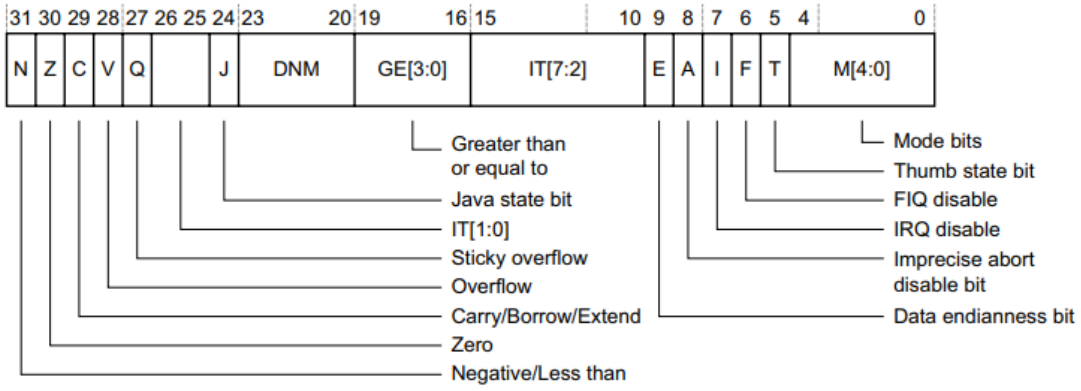

Figura 3.4: Bits de configuração do registrador CPSR [ARM10].

3.2.1 System Control Coprocessor

A arquitetura ARMv7 utiliza também um *System Control Coprocessor* (CP15) para permitir o controlar as funções do processador. O CP15 possui um conjunto de registradores para ler e escrever nas funções que o processador irá realizar [ARM10]. O CP15 é configurável de acordo com a extensão de segurança TrustZone. Alguns registradores são acessíveis apenas no modo seguro, outros sendo isolados em ambos os modos, e alguns sendo comuns para ambos os modos.

De acordo com o *Cortex-A8 Technical Refence Manual* [ARM10], esses registradores podem ser divididos em 6 grupos funcionais:

- Controle e configuração do sistema;
- Controle e configuração da MMU;
- Controle e configuração da cache;
- Controle e configuração da *L2 cache preload engine*;
- System performance monitor;
- *Array debug*, habilita a depuração do processador Cortex-A8 acessando dados apenas em um estado seguro.

Para configurar o *System Control Coprocessor* (CP15) é necessário escrever em seus registradores. Para isso são utilizados as instruções MRS e MSR. A instrução MSR é utilizada para transferir uma informação contida em um registrador ARM para um registrador do CP15 [ARM11]. Abaixo pode ser vista a sintaxe dessa instrução:

```
MSR{cond} coproc_register, Rn       // {cond} -  optional condition code.
```

Exemplo:

```
MSR SCTLR, R1 ; writes the contents of R1 into the CP15 coprocessor register SCTLR
```

A instrução MRS é utilizada para transferir uma informação contida em um registrador do CP15 para um registrador do ARM [ARM11]. Abaixo pode ser vista a sintaxe dessa instrução:

```
MRS{cond} Rd, psr
```

Exemplo:

```
MRS R1, SCTLR ; writes the contents of the CP15 coprocessor register SCTLR into R1
```

O artigo *TrustZone Explained: Architectural Features and Use Cases* [NMB+16] destaca a utilização dos seguintes registradores do CP15:

- Vector Base Address Register (VBAR) - Registrador que contém o endereço base do vetor de interrupções, tanto o modo seguro quando o não seguro, que não são gerenciadas pelo modo de operação monitor (os modos de operação do processador são descritos posteriormente, na Tabela 3.2). Esse registrador é configurado de forma independente para o modo seguro e o modo não seguro [ARM10];
- Monitor Vector Base Address Register (MVBAR) - Registrador que contém o endereço base do vetor de interrupções para as exceções do modo monitor. O MVBAR permite a escrita e leitura somente no modo seguro [ARM10];
- Interrupt Status Register (ISR) - Exibe o estado das das interrupções IRQ, FIQ e SError [ARM10];
- Secure Configuration Register (SCR) - Utilizado para definir o estado do processador, não seguro ou seguro, o estado em que as exceções serão executadas e modificar os bits A até I do CPSR quando o processador está no estado não seguro. A escrita nesse registrador só é permitida em modo seguro [ARM10];
- Translation Table Base Register (TTBR0 e TTBR1) - Registradores que armazenam a *translation table* (TLB) [ARM10];
- Translation Table Base Control Register (TTBCR) - Utilizado para realizar configurações na TLB [ARM10];
- Domain Access Control Register (DACR) - Registrador que configura a permissão de acesso para até 16 domínios. Estes domínios são endereçamentos de memória [ARM10];
- Non-Secure Access Control Register (NSACR) - Responsável por definir as permissões de acesso não-seguras para os coprocessadores e para a preload engine (PLE) [ARM10];
- Secure Debug Enable Register (SDER) - Registrador utilizado para prover o controle de permissões de *debug* para o modo de usuário seguro [ARM10].

3.2.2 Exceções

A arquitetura ARMv7-A possui um *vector table* para suas exceções. Essa tabela corresponde a um conjunto de endereços, os quais contém os endereços para as funções

de tratamento de exceção. Quando uma exceção ocorre, o *Program Counter*(PC) é movido para o endereço referente ao tipo de exceção ocorrida. Esse endereço é chamado de vetor de exceção para aquela exceção [ARM14].

Tabela 3.1: *Vector table* e suas descrição.

Offset	Execeção	Descrição
0x00	Reset	-
0x04	Undefined Instruction	Ocorre quando o processador tenta executar uma instrução inválida [ARM19].
0x08	SMC / SVC	Executada quando uma exceção de chamada do secure monitor (SMC) ou do supervisor (SVC) é realizada [ARM19].
0x10	Data Abort	Ocorre quando uma função de transferência de dados tenta carregar ou armazenar um dado de um endereço inválido [ARM19].
0x14	Not used	-
0x18	IRQ interrupt	O IRQ é uma interrupção causada por outro hardware quando ele deseja notificar o processador [ARM19].
0x1C	FIQ interrupt	O FIQ é uma interrupção com o mesmo propósito do IRQ. A diferença é que o FIQ possui uma prioridade superior no processador [ARM19].

O Cortex-A8 possui até 4 *vector tables*, sendo eles o seguro e não seguro *Vector Base Address Register*, o *monitor vector base address register* e o *Hyp Vector Base Address Register* [ARM10].

O Vector Base Address Register pode ser configurado para alterar o endereço do vetor de interrupções, o seu endereço de reset é o 0x0. O código 3.1, retirado do livro ARM Compiler Software Development Guide [ARM14], mostra um exemplo simples de um Vector Base Address alinhado no endereço de reset 0x0.

Código 3.1: Exemplo de vector table [ARM14].

```
1                AREA vectors, CODE, READONLY
2                ENTRY
3  Vector_Table
4                LDR pc, Reset_Addr
5                LDR pc, Undefined_Addr
6                LDR pc, SVC_Addr
7                LDR pc, Prefetch_Addr
```

```
8                LDR pc, Abort_Addr
9                NOP                     ;Reserved vector
10               LDR pc, IRQ_Addr
11   FIQ_Handler
12               ; FIQ handler code - max 4kB in size
13   Reset_Addr      DCD Reset_Handler    ; DCD:  allocates one or more words of memory,
     ↪  aligned on four-byte boundaries
14   Undefined_Addr  DCD Undefined_Handler
15   SVC_Addr        DCD SVC_Handler
16   Prefetch_Addr   DCD Prefetch_Handler
17   Abort_Addr      DCD Abort_Handler
18   IRQ_Addr        DCD IRQ_Handler
19               ...
20               END
```

Para habilitar as interrupções IRQ e FIQ, os bits 6 e 7 do registrador CPSR devem ser escritos. O código 3.2 habilita as interrupções IRQ e FIQ escrevendo 0xC0 no CPSR. Para desabilitar as interrupções basta negar os bits escritos no código 3.2.

Código 3.2: Código para habilitar as interrupções IRQ e FIQ.

```
1   MRS     R0, CPSR
2   BIC     R0, R0, 0xC0   ;   BIC:  Bit Clear
3   MSR     CPSR, R0
```

3.2.3 Monitor e Modos de Operação do ARMv7

A arquitetura ARMv7 possui 8 modos de operação. Para alternar entre os modos de operação é necessário escrever nos bits 4 a 0 do CPSC, como pode ser visto na figura 3.4. A tabela 3.2 lista e descreve cada um dos modos de operação com a extensão TrustZone habilitada. A coluna código é o valor que deve ser escrito nos bits descritos previamente do CPSC.

3.2.4 Manipulando os modos de segurança no Cortex-A8

Para alternar os modos de segurança do processador é necessário escrever no non-secure (NS) bit do Secure Configuration Register (SCR). O código 3.3 é um exemplo baseado no código do artigo *Imperas ARM TrustZone Platform Modeling Application Note* [Imp18] que lê e escreve no NS bit. Como pode ser visto, o define WR_SCR irá colocar o

Tabela 3.2: Modos de Operação do ARMv7.

Modo	Código	Descrição
User	10000	Um sistema operacional executa normalmente as suas aplicações nesse modo para restringir o uso dos recursos do sistema. O modo de usuário somente realiza acessos sem privilégios e não pode acessar recursos protegidos. Esse modo pode ser executado no estado seguro e não seguro [ARM10].
FIQ	10001	Modo de operação que o processador entra quando uma interrupção FIQ ocorre. Esse modo pode ser executado tanto no modo seguro quanto no não seguro [ARM10].
IRQ	10010	Modo de operação que o processador entra quando uma interrupção IRQ ocorre. Esse modo pode ser executado tanto no modo seguro quanto no não seguro [ARM10].
Supervisor	10011	Modo de operação que o processador entra quando uma Supervisor Call ocorre. O processador é inicializado no reset em modo supervisor. Esse modo pode ser executado tanto no modo seguro quanto no não seguro [ARM10].
Monitor	10101	Modo de operação que o processador entra quando uma Secure Monitor Call (SMC) ocorre. Esse modo somente é executado em modo seguro e sempre irá entrar em modo seguro quando uma SMC for executada [ARM10].
Abort	10111	Modo de operação que é adotado pelo processador quando uma exceção relacionada a memória ocorre [ARM10].
Undefined	11011	Modo de operação que é adotada pelo processador quando uma execução relacionada a instruções inválida ocorre [ARM10].
System	11111	Modo de execução com os mesmo privilégios do modo *User* [ARM10].

valor da variável _VAL no registrador SCR. Ao final do exemplo, na linha 13, é escrito 1 no SCR. Como o NS bit é o bit menos significativo do registrador, escrever somente 0x01 nele irá ativar o bit. O NS bit desativa o modo seguro quando escrito 1.

Código 3.3: Exemplo de código que escreve no NS bit.

```
1 // CP15 register access
2 #define WR_CP15(_OP1, _CRN, _CRM, _OP2, _VAL) \
3     asm volatile("mcr p15, "#_OP1", %0, "#_CRN", "#_CRM", "#_OP2::"r"(_VAL))
4
5 #define RD_CP15(_OP1, _CRN, _CRM, _OP2, _VAL) \
6     asm volatile("mrc p15, "#_OP1", %0, "#_CRN", "#_CRM", "#_OP2:"=r"(_VAL))
7
8 #define WR_SCR(_VAL)   WR_CP15(0, c1, c1, 0, _VAL)
```

```
#define RD_SCR(_VAL)   RD_CP15(0, c1, c1, 0, _VAL)

...
//enter non-secure
WR_SCR(1);
```

No livro *Cortex-A8 techinical Reference Manual*, capítulo 2.16 - *Software consideration for Security Exentensions*[ARM10], é dito que o modo de operação monitor deve ser utilizado para trocar os estados de segurança do processador. Para realizar uma troca de contexto entre o modo seguro e o não seguro, uma exceção deve ser gerada através da instrução SMC e então a troca de modo de segurança deve ser feita. O whitepaper *SMC Calling Conventions* [ARM16] sugere utilizar os registradores R1-R5 para passar argumentos ao código monitor. O código 3.4 é um exemplo de monitor que irá tratar o acesso ao modo seguro de acordo com o argumento passado pelo registrador r2.

Código 3.4: Código de tratamento de uma SMC.

```
register r2 asm("r2");
void SMC_Handler_Main( )
{
    int smc_number = r2;
    printf("NS - Entered in Super Monitor Call Handler!!! Operation is %d\n",
    ↪ smc_number);
    switch (smc_number)
    {
    case TO_SECURE:
        printf("Writing 0x0 to ns bit\n");
        WR_SCR(0);
        break;
    case TO_UNSECURE:
        printf("Writing 1 to ns bit\n");
        WR_SCR(1);
        break;
    default:
        asm(
            "MSR SPSR_cxsf, #0x13\n"
            "LDR PC, =run\n" // SVC_entry points to the first
        );
        break;
    }
}

int main()
```

```
26 {
27 ...
28 // enter in non-secure mode
29 asm("mov r2, #2\n"
30     "SMC #0\n")
31 ...
32 }
```

4. IMPLEMENTAÇÃO E VALIDAÇÃO

Esse Capítulo apresenta a implementação de um sistema de comunicação entre processadores ARM utilizando a extensão de segurança TrustZone.

Para validar a comunicação entre processadores foi criado uma plataforma do sistema utilizando o *Open Virtual Platform* (OVP), ferramenta previamente descrita nesse documento. Na ferramenta OVP foi projetado um periférico denominado **NonSecToSec**. Esse periférico é responsável por transmitir mensagens da região de processadores não seguros (**RNS**) para o processador da região segura (**RS**).

A plataforma de validação é composta por dois processadores Cortex-A8 na região não segura, cada um com sua respectiva memória, representando sistemas que enviam dados sigilosos oriundos de fontes passíveis a ataques, como por exemplo dados de sensores ou da rede. O sistema também contém o periférico *NonSecToSec*, um processador Cortex-A8 na região segura, uma memória não segura para propósito geral, uma memória segura para dados sigilosos, e o periférico TZPC para controlar os acessos ao periférico NonSecToSec. A Figura 4.1 apresenta a plataforma de validação criada.

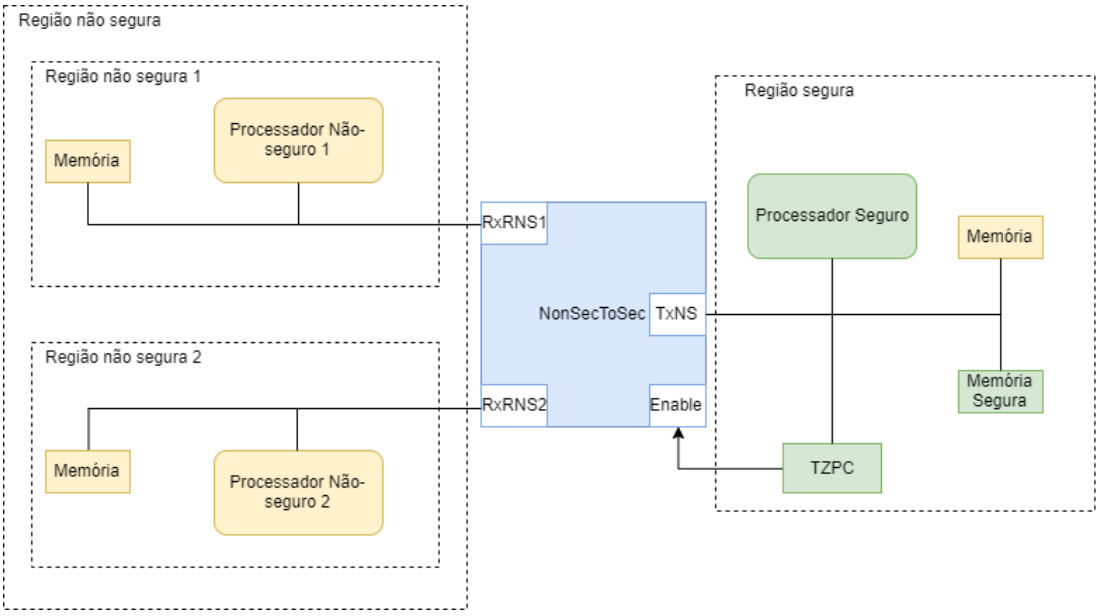

Figura 4.1: Plataforma de validação para o sistema de comunicação entre processadores.

O periférico *NonSecToSec* realiza o controle dos processadores que querem enviar dados para a região segura. O periférico possui um buffer circular e permite o envio de uma mensagem por vez para o processador seguro. Quando um processador não seguro envia sua informação para o periférico, ele irá gerar uma interrupção no processador seguro. O processador seguro verifica qual processador enviou a informação, transaciona para o modo seguro, e armazena a informação enviada na memória protegida. Esta memória está mapeada em um endereço que somente é acessada pelo modo seguro, realizando

como já descrito previamente nesse documento, uma exceção de acesso sempre quando um acesso não seguro for realizado a essa memória.

O código fonte, em linguagem TCL, para modelar esta plataforma é apresentada no **Apêndice A**.

4.1 Periférico NonSecToSec

O periférico *NonSecToSec* possui três barramentos para transmissão de dados, dois para a região não segura (processadores que enviam mensagem para a região segura) e uma para a região segura (transmite as informações da região não segura para a segura). O periférico possui uma porta *enable*, ligada ao periférico TZPC, e uma porta que é habilitada quando um novo dado está pronto para ser enviado à região segura. Maiores detalhes sobre as portas podem se encontradas na tabela 4.1.

Tabela 4.1: Portas e Barramentos do Periférico NonSecToSec.

Nome	Tipo	Descrição
enable	netport de entrada	Porta de 1 bit que habilita a transferência de mensagens da região não segura para a região segura.
interruptRequest	netport de saída	Porta de 1 bit que fica em nível lógico alto quando uma nova mensagem está disponível para ser enviada para a região segura.
rxNonSecurePort1	barramento 32 bits	Esse barramento é responsável por receber as mensagens de um processador não seguro. Ele possui registradores para ser requisitada a escrita, avisar a permissão da escrita e transferir dados
rxNonSecurePort2	barramento 32 bits	Barramento com o mesmo propósito do rxNonSecurePort1.
readSecurePort	barramento 32 bits	Barramento responsável por transferir as mensagens para o processador seguro. Ele possui registradores para implementar o protocolo de comunicação e transferir os dados.

O código fonte, em linguagem TCL, para modelar este periférico é apresentado no **Apêndice B**.

Este periférico implementa dois protocolos de comunicação, um para o recebimento de dados da região não segura e outro para enviar dados à região segura.

4.1.1 Protocolo de recebimento de dados da região não segura

Para iniciar um envio, deve-se escrever um bit em nível lógico alto no registrador de requisição de uma das portas rxNonSecurePort, mapeado com o nome rx_req, e então deve ser verificado se o registrador rx_ack está em nível lógico alto. O periférico *NonSecToSec* irá apenas habilitar o registrador rx_ack se nenhum outro processador está realizando uma comunicação com a região segura. Este mecanismo é utilizado para garantir robustez quando múltiplos processadores tentam enviar dados à região segura. Se o rx_ack estiver em nível lógico 1, é possível escrever na região do cabeçalho do barramento (rx_header). Este cabeçalho espera que o processador identifique-se e defina o o tamanho da mensagem que irá enviar. Após o periférico conhecer o tamanho da mensagem, ele aloca uma memória interna para receber a mensagem. O último passo é escrever a mensagem no registrador rx_data conforme o tamanho especificado no cabeçalho. O periférico não irá receber outra mensagem até que o dado enviado pelo processador não seguro seja lido completamente.

O código 4.1 implementa a funcionalidade que garante que apenas um processador escreva por vez no periférico *NonSecToSec*. A função *canGivenRNSTransmit* será chamada sempre que um processador escrever no registrador de requisição de envio. Esta função recebe por parâmetro o ID desse processador. Na linha 20 o método *isSomeRNSTransmitingData* irá verificar se algum processador da região não segura (RNS) está transmitindo dados. Quando nenhum processador está transmitindo dados, as linhas 20 e 21 alocam o envio de mensagens somente para a porta que escreveu nesse momento no periférico.

Código 4.1: Tratamento de Múltiplos Processadores requisitando envio de dados.

```
1  #define NUMBER_OF_RNS 2
2  Bool isSomeRNSTransmitingData()
3  {
4      Bool ret = False;
5      int i = 0;
6
7      for(i = 0; i < NUMBER_OF_RNS; i++)
8      {
9       if(rnsRequest[i] == TRANSMITTING)
10         {
11             ret = True;
12         }
13     }
14     return ret;
15 }
```

```
16
17 Bool canGivenRNSTransmit(int cpuRNSId)
18 {
19     Bool ret = False;
20     if(isSomeRNSTransmitingData() == False)
21     {
22         ret = True;
23         rnsRequest[cpuRNSId] = TRANSMITTING;
24         bhmMessage("I","RNS TX Register","RNS%d is now transmitting!", cpuRNSId+1 );
25     }
26     return ret;
27 }
```

4.1.2 Protocolo de envio de dados para a região segura

Após um processador da região não segura terminar de escrever no buffer alocado para a sua mensagem, o *NonSecToSec* irá colocar a porta *newMessageAvailable* em nível lógico alto até o momento em que o registrador *ns_interrupt_ack* do barramento *readSecurePort* receba um bit em 1, conforme pode ser visto nas linhas 1 e 5 do código 4.2. Após, a porta *enable* deve ser habilitada para permitir a leitura dos dados no barramento *readSecurePort*, comportamento que é realizado pelo TZPC na plataforma implementada.

O dado fica disponível até ele ser completamente lido. Após, o periférico irá permitir que outro processador da região não segura envie novos dados, seguindo os passos descritos para o envio de dados da região não segura.

O comportamento implementado de leitura de dados feito no periférico pode ser visto na função *PPM_READ_CB(dataRead)* do código 4.2. Na linha 11 é feita a verificação se a transmissão foi habilitada. A leitura dos dados no buffer do periférico é realizado nas linha 17 e 20. O periférico libera a transmissão para novos dados nas linhas 26, 27 e 28, depois que toda a informação do seu buffer foi consumida.

Código 4.2: Tratamento das portas de ACK da interrupção e envio de dados para a região segura.

```
1 PPM_REG_WRITE_CB(interruptWrite)
2 {
3     bhmMessage("I","Interrupt","Received Interrupt ACK");
4     ppmWriteNet(handles.newMessageAvailable, 0);
5 }
6 ...
7 PPM_READ_CB(dataRead)
```

```
{
    static int dataReadCount = 0;
    char rtValue = 0;
    if(transmissionEnabled)
    {
        if(dataReadCount < header.messageSize)
        {
            bhmMessage("I","Secure buffer data","Reading");
            char* prtValue = 0;
            prtValue =
            ↪  &((char*)handles.readSecurePort)[READSECUREPORT_REGS_NS_DATA_OFFSET];
            prtValue = prtValue + dataReadCount;
            rtValue = *prtValue;
            dataReadCount++;
        }

        if(dataReadCount >= header.messageSize)
        {
            bhmMessage("I","Secure buffer data","No message in buffer, releasing to
            ↪  any port write");
            rnsRequest[0] = FREE;
            rnsRequest[1] = FREE;
            dataReadCount = 0;
        }
    }
    else
    {
        bhmMessage("I","Secure buffer data","data transfer is disabled");
    }

    return rtValue;
}
```

4.2 Ambiente não seguro

Para implementar o protocolo de envio de dados, duas funções foram criadas para serem utilizadas por ambos processadores não seguros: **requireToSend** e o **sendMessage**. A função **requireToSend** é bloqueante. Ela irá escrever no registrador rx_req e aguardar até o periférico *NonSecToSec* permitir que ele envie um dado. A função **sendMessage** irá escrever o cabeçalho da mensagem e escrever no registrador do periférico os

dados a serem enviados para a região segura. A implementação dessas funções é apresenta no código 4.3.

Código 4.3: Funções de envio de dados.

```
1  // @brief: Require permission to peripheral to send a message.
2  // Blocking method.
3  void requireToSend()
4  {
5      printf("RNS: Requiring write permission\n");
6      (*(char*) RG_TX_REQ) = 1;
7      while(*RG_TX_REQ_ACK != 1);
8      printf("RNS: Ready to send!\n");
9  }
10
11 // @brief: Send given message filling the header with parameter sender and size.
12 void sendMessage(int size, int sender,char* data)
13 {
14     *RG_TX_WRITE_HEADER = sender;
15
16     *RG_TX_WRITE_HEADER = size;
17
18     int i = 0;
19     for(i = 0; i < size; i++ )
20     {
21        *RG_TX_WRITE_DATA = data[i];
22     }
23     printf("RNS: Sent a message!\n");
24 }
```

Os dois processadores não seguros fazem uso das funções descritas acima e enviam dados para o periférico *NonSecToSec*.

4.3 Ambiente seguro

O software do processador seguro, diferente dos não seguros, precisa habilitar interrupções, permitir o transacionamento dos modos seguros e não seguros, mapear a memória ao periférico TZPC e escrever a *vector table*.

O processador seguro possui três principais blocos de código: *boot*, *run* e tratamento de interrupções:

- *boot* - realiza a configuração das interrupções, mapeia a Translation Tookside Table (TLB) e as instruções da vector table partindo do endereço 0x0. Essas configurações são feitas conforme exemplificado no capítulo 3.
- *run* - é a execução do cálculo de fibonnati, simulando uma operação comum que o processador realizaria em seu modo de operação de usuário e não seguro.
- tratamento de interrupções - implementa o tratamento da interrupção IRQ que é gerada pelo periférico *NonSecToSec*. Esse código implementa o diagrama de sequência representado na figura 4.2.

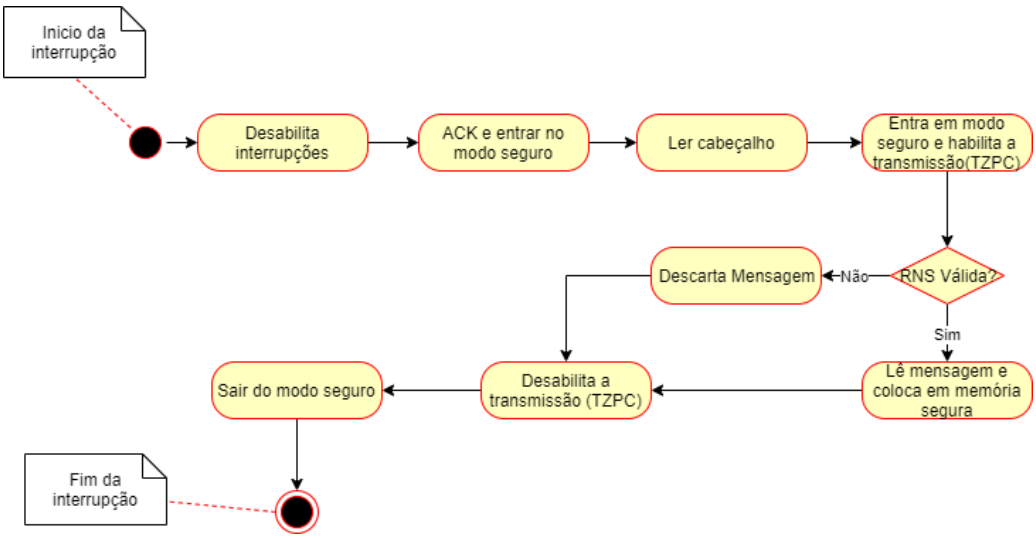

Figura 4.2: Diagrama de atividade da interrupção no processador seguro

Para entrar em modo seguro é executada uma instrução SMC e um código de tratamento similar ao código 3.4, o qual troca o modo de segurança do processador.

Como pode ser visto no código 4.4 de tratamento da interrupção IRQ, o processador seguro irá desabilitar as interrupções (linha 8) e responder com um ACK ao periférico (linha 9) e entrar no modo seguro (linha 13), ler o cabeçalho da mensagem e caso seja uma mensagem de um identificador conhecido, recebê-la e gravar na memória segura. Por fim, a comunicação é desabilitada na linha 28 e o processador volta ao estado não seguro.

Código 4.4: Tratamento da interrupção do processador seguro.

```
1  static void irq_handler(void *ud)
2  {
3      char buffer[HEADER_SIZE]; // buffer to retrieve header information
4      int i; // loop counter
5      messageHeader *header; //message header
6
7      printf("Received interrupt. Starting handle\n\n############\n\n");
```

```
8       DISABLE_INTERRUPTS();
9       ackInterruptHandler();
10
11      printf("Entering in secure world\n");
12
13      ENTER_SECURE_USING_MONITOR();
14
15      printf("Entered in secure world\n");
16
17      for(i = 0; i< HEADER_SIZE; i++)
18      {
19          buffer[i] = *(RG_READ_HEADER_DATA);
20      }
21
22      memcpy(header,buffer,HEADER_SIZE);
23
24      if(header->sender == 0 || header->sender == 1)
25      {
26          ENABLE_BRIDGE_COMMUNICATION();
27          receiveMessage(header->messageSize);
28          DISABLE_BRIDGE_COMMUNICATION();
29      }
30  ...
31      ENTER_NON_SECURE_USING_MONITOR();
32      Exit_irq();
33  }
```

O trecho de código 4.5, mostra o mapeamento em memória representado no código para a localização dos registradores de comunicação do *NonSecToSec* para o processador seguro e a posição da região segura de memória.

Código 4.5: Definições de endereços do processador seguro.

```
1  #define RG_ACK_INTERRUPT ((volatile unsigned char *) (0x11001000))
2  #define RG_READ_HEADER_DATA ((volatile unsigned char *) (0x11001002))
3  #define RG_READ_DATA ((volatile unsigned char *) (0x11001003))
4  #define SECURE_MEMORY_REGION ((volatile unsigned char *) (0x01000000))
```

A função de recebimento dos dados pode ser vista no código 4.6. A linha 10 recebe os dados em uma memória ainda na região não segura e na linha 16 ela é transferida para a região segura. Como pode ser visto, ao final, para não ser possível ler essa memória em outro momento, a função *memset* é chamada para limpar o buffer alocado na memória não segura.

Código 4.6: Leitura dos dados do periférico NonSecToSec.

```
// @brief: Receive a message with given size from peripheral NonSecToSec
// and save it to secure memory.
void receiveMessage(int size)
{
    int i;
    char* nonSecureMemoryBuffer = malloc(size);
    printf("Reading message of %d\n", size);
    printf("####\nRECEIVED WAS:\n");
    for(i = 0; i< size; i++)
    {
        secureMemoryBuffer[i] = *(RG_READ_DATA);
        printf("%c",secureMemoryBuffer[i]);
    }

    printf("\n######\n");
    memcpy(SECURE_MEMORY_REGION,secureMemoryBuffer, size);
    memset(secureMemoryBuffer,0, size);
}
```

4.4 Validação

Para realizar a validação da plataforma, os processadores não seguros enviam duas mensagens para o periférico *NonSecToSec*.

Observa-se na figura 4.3, na etapa 1, que o RNS2 realiza a sua escrita no periférico. A etapa 2 demonstra o *NonSecToSec* negando a segunda escrita do RNS2 assim como a escrita do RNS1. Na etapa 3 pode ser visto que o processador seguro recebe a interrupção gerada pelo periférico.

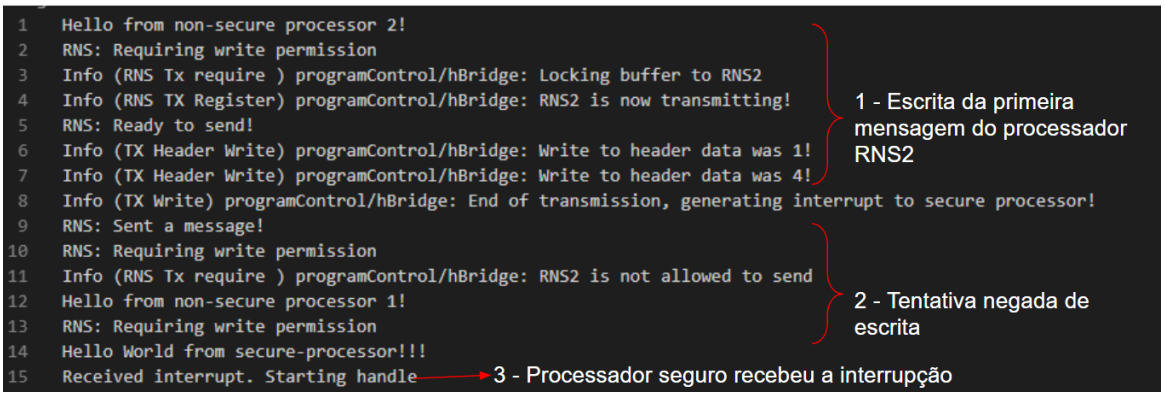

Figura 4.3: Primeira etapa do log da execução

Na etapa 4, na figura 4.4, o processador seguro responde à interrupção com um ACK e então troca para o modo seguro e habilita a transmissão, conforme exemplificado nas etapas 5 e 6. A etapa 7 e 8 é a leitura da mensagem e, por fim, na etapa 9 o processador termina a interrupção e volta ao modo não seguro.

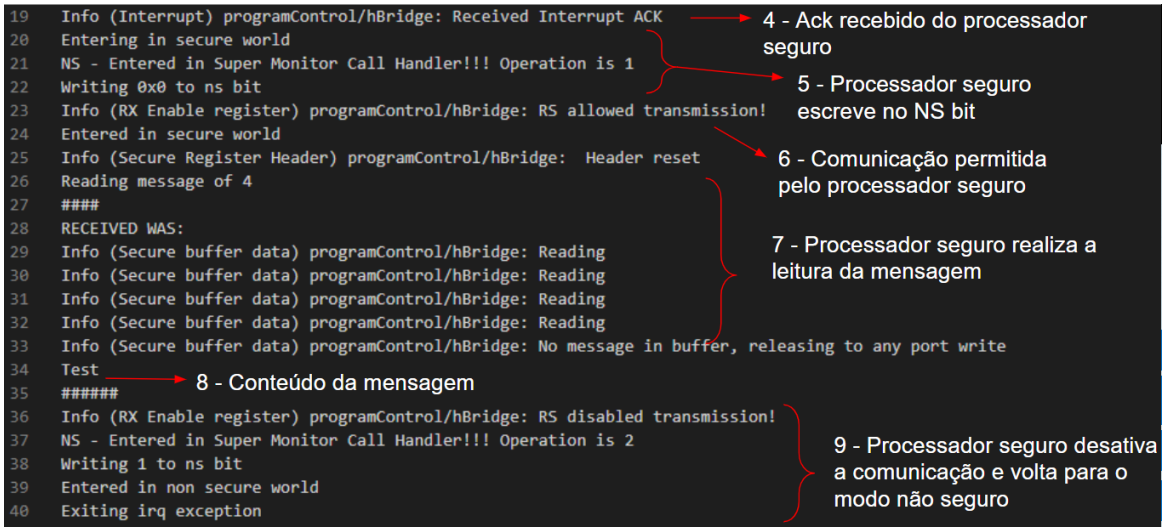

Figura 4.4: Secunda etapa do log da execução

Esse mesmo processo ocorre para as outras três mensagens enviadas pelos processadores não seguros. Ao final da validação, o processador seguro exibe o primeiro caractere na memória segura no modo seguro e tenta exibir o segundo caractere no modo não seguro. Como pode ser visto na figura 4.5, o acesso não seguro à memória gera uma exceção de acesso no processador.

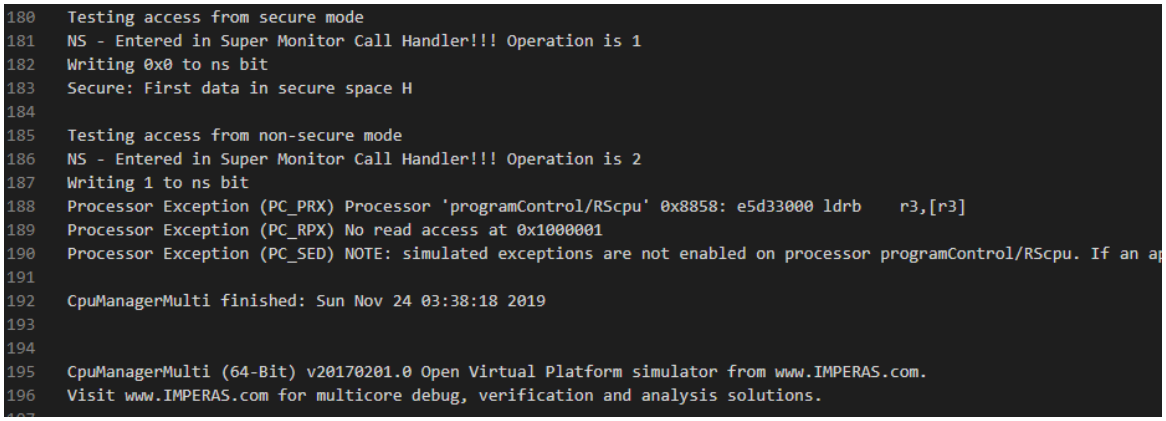

Figura 4.5: Teste de acesso aos dados em memória segura.

Conforme demonstrado na plataforma de validação criada, utilizar a extensão de segurança TrustZone implementa duas propriedades de segurança: integridade e disponibilidade. As duas propriedades são implementadas no sistema seguro ao realizar o isolamento do sistema em duas partições, uma segura e uma não segura. A comunicação das mensagens são em claro, já que, a extensão TrustZone permite a implementação de um

sistema de isolamento e não meios para a troca de informações entre processadores. Assim como foi concluído por Ngabonziza et al. [NMB+16], a tecnologia TrustZone não provê uma segurança nativa ao sistema e sim deve ser implementada junto de outros hardwares e software.

5. CONCLUSÃO

O presente trabalho de conclusão possibilitou agregar conhecimentos na área de arquitetura de computadores através dos estudos da arquitetura do processador ARM, na área de sistemas operacionais, através da codificação de um sistema básico em *bare metal* e, por fim, na área microcontroladores com as simulações na plataforma OVP e com o estudo da arquitetura do processador ARM. O trabalho permitiu a experiência de programação em uma arquitetura de processador relevante para a indústria, assim como práticas relevantes para a implementação de *firmwares*.

Este trabalho mostrou um SoC simulado para realizar a comunicação entre processadores sem mecanismos de segurança para um processador seguro através de um periférico modelado na plataforma OVP. Um protocolo foi criado para gerir as múltiplas mensagens enviadas dos processadores não seguros para o processador seguro.

A plataforma de validação final pôde demonstrar que o uso da extensão de segurança TrustZone permite a criação de um ambiente de execução seguro e proteção de acesso à memória segura no processador, porém transmitindo as mensagens ao periférico criado em claro.

Na pesquisa dos mecanismos de configuração do processador ARM, assim como na implementação de como gerar as interrupções e exceções do mesmo, demandou um alto esforço de pesquisa, principalmente quando utilizando os mecanismos da extensão de segurança TrustZone.

A plataforma criada nesse trabalho e o software do processador seguro podem ser aprimorados em trabalhos futuros. Na questão da segurança da informação, o protocolo utilizado pelo periférico poderia utilizar um mecanismo para garantir a integridade das informações, criptografia poderia ser aplicada às mensagens e mecanismos de autenticação poderiam ser adicionados aos protocolos de comunicação. Além disso, o sistema de controle de chaveamento de contexto poderia ser aprimorado no sistema do processador seguro.

REFERÊNCIAS BIBLIOGRÁFICAS

[ARM09] ARM. “ARM Security Technology Building a Secure System using TrustZone Technology”, 2009, Capturado em: https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf?_ga=2.36177632.1764020838.1568651964-1471959229.1568651964.

[ARM10] ARM. “Cortex-A8 - Technical Reference Manual”, 2010, Capturado em: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf.

[ARM11] ARM. “ARM Compiler toolchain”, 2011, Capturado em: http://infocenter.arm.com/help/topic/com.arm.doc.dui0489c/DUI0489C_arm_assembler_reference.pdf.

[ARM14] ARM. “ARM Compiler Software Development Guide”, 2014, Capturado em: http://infocenter.arm.com/help/topic/com.arm.doc.dui0471k/DUI0471K_software_development_guide.pdf.

[ARM16] ARM. “SMC CALLING CONVENTION - System Software on ARM Platforms”, 2016, Capturado em: https://chasinglulu.github.io/downloads/ARM_DEN0028B_SMC_Calling_Convention.pdf.

[ARM19] ARM. “Arm Architecture Reference Manual - Armv8, for Armv8-A architecture profile”, 2019, Capturado em: https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.153466424.1474128376.1574655536-220125155.1571781081.

[Bak16] Bakos, J. D. “Embedded Systems - ARM Programming and Optimization”. Morgan Kaufmann, 2016, 320p.

[Imp18] Imperas. “Imperas ARM TrustZone Platform Modeling Application Note”, 2018, Capturado em: http://www.imperas.com/documents/Imperas_ARM_TrustZone_Platform_Modeling_Application_Note.pdf.

[Imp19a] Imperas. “iGen Platform and Module Creation User Guide”, 2019, Capturado em: http://www.ovpworld.org/documents/iGen_Platform_and_Module_Creation_User_Guide.pdf.

[IMP19b] IMPERAS. “Open Virtual Plataforms website”. Capturado em: http://www.ovpworld.org, 2019.

[NMB+16] Ngabonziza, B.; Martin, D.; Bailey, A.; Cho, H.; Martin, S. “TrustZone Explained: Architectural Features and Use Cases”. In: IEEE International Conference on Collaboration and Internet Computing CIC, 2016, pp. 445–451.

[PP15] Pfleeger, C. P.; Pfleeger, S. L. “Security in Computing”. Prentice Hall Professional Technical Reference, 2015, 5th ed., 944p.

APÊNDICE A – DESCRIÇÃO DA PLATAFORMA EM TCL

Código A.1: Código fonte da plataforma de hardware.

```
ihwnew -name programControl
ihwaddclp -allargs

ihwaddbus -instancename pBusRS  -addresswidth 32
# tzBus - 41 bits of address - implements secure and non-secure address spaces
ihwaddbus -instancename tzBus -addresswidth 41

## Region secure CPU
ihwaddprocessor -instancename RScpu -type arm -vendor arm.ovpworld.org -endian little
  ↪ -semihostname armNewlib -semihostvendor arm.ovpworld.org

ihwconnect      -instancename RScpu -busmasterport INSTRUCTION -bus tzBus
ihwconnect      -instancename RScpu -busmasterport DATA        -bus tzBus
ihwconnect      -instancename RScpu -netport irq -net fiq_

ihwsetparameter   -handle       RScpu -name variant       -value Cortex-A8   -type
  ↪ enum

ihwaddmemory -instancename ramRS -type ram
ihwconnect   -instancename ramRS -busslaveport sp1 -bus pBusRS -loaddress 0x00000000
  ↪ -hiaddress 0x001fffff

ihwaddmemory -instancename secureRam -type ram
ihwconnect   -instancename secureRam -busslaveport sp1 -bus pBusRS -loaddress
  ↪ 0x01000000 -hiaddress 0x01ffffff

#
# Bridge - Secure space of tzBus always mapped to pBus
#
ihwaddbridge -instancename secure2
ihwconnect   -instancename secure2 -busslaveport  sp -bus tzBus -loaddress 0x00000000
  ↪ -hiaddress 0xffffffff
ihwconnect   -instancename secure2 -busmasterport mp -bus pBusRS  -loaddress
  ↪ 0x00000000 -hiaddress 0xffffffff

#
# Bridge - Memory always accessible from non-secure mode
```

```
32 #
33 ihwaddbridge -instancename ram0NS
34 ihwconnect   -instancename ram0NS -busslaveport  sp -bus tzBus -loaddress
   ↪ 0x10000000000 -hiaddress 0x100001fffff
35 ihwconnect   -instancename ram0NS -busmasterport mp -bus pBusRS -loaddress 0x00000000
   ↪ -hiaddress 0x001fffff
36
37 #
38 #-loaddress 0x11001000 -hiaddress 0x11001007
39 ihwaddbridge -instancename hBNSRS
40 ihwconnect   -instancename hBNSRS -busslaveport  sp -bus tzBus -loaddress
   ↪ 0x10011001000 -hiaddress 0x10011001002
41 ihwconnect   -instancename hBNSRS -busmasterport mp -bus pBusRS -loaddress 0x11001000
   ↪ -hiaddress 0x11001002
42
43 #
44 # TZPC: 0x10001000    0x10001fff  BP147 TrustZone Protection Controller
45 #
46 ihwaddperipheral -instancename tzpc -type TzpcBP147  -vendor arm.ovpworld.org
47 ihwconnect       -instancename tzpc -busslaveport bport1 -bus pBusRS -loaddress
   ↪ 0x10001000 -hiaddress 0x10001fff
48 ihwconnect       -instancename tzpc -netport TZPCDECPROT0_0 -net tzpcdecprot0_0
49
50
51 # Non secure processor 1
52
53 ihwaddbus -instancename pBus  -addresswidth 32
54
55 ihwaddprocessor -instancename cpuRNS1 -type arm -vendor arm.ovpworld.org -endian
   ↪ little  -semihostname armNewlib -semihostvendor arm.ovpworld.org
56
57 ihwconnect      -instancename cpuRNS1 -busmasterport INSTRUCTION -bus pBus
58 ihwconnect      -instancename cpuRNS1 -busmasterport DATA        -bus pBus
59
60 ihwsetparameter   -handle       cpuRNS1 -name variant       -value Cortex-A8   -type
   ↪ enum
61
62 ihwaddmemory -instancename ramRNS1 -type ram
63 ihwconnect   -instancename ramRNS1 -busslaveport sp1 -bus pBus -loaddress 0x00000000
   ↪ -hiaddress 0x001fffff
64
65
```

```
66 # Non secure processor 2
67
68 ihwaddbus -instancename pBusRNS2  -addresswidth 32
69
70 ihwaddprocessor -instancename cpuRNS2 -type arm -vendor arm.ovpworld.org -endian
   ↪ little -semihostname armNewlib -semihostvendor arm.ovpworld.org
71
72 ihwconnect      -instancename cpuRNS2 -busmasterport INSTRUCTION -bus pBusRNS2
73 ihwconnect      -instancename cpuRNS2 -busmasterport DATA        -bus pBusRNS2
74
75 ihwsetparameter   -handle       cpuRNS2 -name variant       -value Cortex-A8   -type
   ↪ enum
76
77 ihwaddmemory -instancename ramRNS2 -type ram
78 ihwconnect   -instancename ramRNS2 -busslaveport sp1 -bus pBusRNS2 -loaddress
   ↪ 0x00000000 -hiaddress 0x001fffff
79
80
81
82 # NonSecToSec
83 ihwaddperipheral -instancename hBridge -type NonSecToSec -modelfile peripheral/pse
84
85 ihwconnect       -instancename hBridge -netport enable -net tzpcdecprot0_0
86 ihwconnect       -instancename hBridge -netport newMessageAvailable -net fiq_
87
88 ihwsetparameter    -handle       hBridge -name portSize    -value        0x1000
   ↪ -type uns64
89
90 ihwconnect      -instancename hBridge -busslaveport readSecurePort -bus pBusRS
   ↪ -loaddress 0x11001000 -hiaddress 0x11001007
91
92 ihwconnect      -instancename hBridge -busslaveport rxNonSecurePort1 -bus pBus
   ↪ -loaddress 0x11001000 -hiaddress 0x1100100b
93
94 ihwconnect      -instancename hBridge -busslaveport rxNonSecurePort2 -bus pBusRNS2
   ↪ -loaddress 0x11001000 -hiaddress 0x1100100b
```

APÊNDICE B – DESCRIÇÃO DO PERIFÉRICO NONSECTOSEC EM TCL

Código B.1: Código fonte da plataforma de hardware do periférico NonSecToSec.

```
1  imodelnewperipheral  -name        NonSecToSec -library     peripheral -version
   ↪  0.1 -constructor constructor -visibility    visible
2
3  # Output bus size is 32 bits
4  set outputBusWidth  32
5
6
7  #GENERAL
8  # Define enable net input port
9  imodeladdnetport -name enable -type input -updatefunction updateEnable
10
11 # Define formals for specifying the output master and slave bus port location and
   ↪  size
12 imodeladdformal -name portSize    -type integer
13 imodeladdformal -name enableBridge -type bool
14
15
16
17 ## Secure registers
18 imodeladdbusslaveport -name readSecurePort -mustbeconnected
19
20 imodeladdaddressblock -name regs   -port readSecurePort -offset 0x0 -width 16 -size 8
21
22 imodeladdmmregister  -name ns_interrupt_ack -addressblock readSecurePort/regs
   ↪  -readfunction   interruptRead -writefunction  interruptWrite
23
24 imodeladdlocalmemory -name ns_data_header -addressblock readSecurePort/regs
   ↪  -readfunction dataHeaderRead -offset 1 -size 2
25
26 imodeladdlocalmemory -name ns_data -addressblock readSecurePort/regs -readfunction
   ↪  dataRead -offset 3 -size 1
27
28 ###
29 imodeladdnetport -name newMessageAvailable -type output
30
31
32 # Non secure registers - RNS1
```

```
33
34 imodeladdbusslaveport -name rxNonSecurePort1
35
36 imodeladdaddressblock -name rxRegs -port rxNonSecurePort1 -offset 0x0 -width 8 -size
   ↪ 12
37
38 imodeladdmmregister  -name rx_req -addressblock rxNonSecurePort1/rxRegs
   ↪ -writefunction writeTxReq -offset 0x0 -access w
39
40 imodeladdmmregister  -name rx_ack -addressblock rxNonSecurePort1/rxRegs -readfunction
   ↪ readTxAck -offset 0x1 -access r
41
42
43 imodeladdmmregister  -name rx_header -addressblock rxNonSecurePort1/rxRegs
   ↪ -writefunction  txWriteHeader -offset 0x2 -width 2 -access w
44
45 imodeladdmmregister  -name rx_data -addressblock rxNonSecurePort1/rxRegs
   ↪ -writefunction  txWrite -offset 0x4 -access w
46
47
48 # Non secure registers - RNS2
49
50 imodeladdbusslaveport -name rxNonSecurePort2
51
52 imodeladdaddressblock -name rxRegs -port rxNonSecurePort2 -offset 0x0 -width 8 -size
   ↪ 12
53
54 imodeladdmmregister  -name rx_req -addressblock rxNonSecurePort2/rxRegs
   ↪ -writefunction writeTxReqRNS2 -offset 0x0 -access w
55
56 imodeladdmmregister  -name rx_ack -addressblock rxNonSecurePort2/rxRegs -readfunction
   ↪ readTxAckRNS2 -offset 0x1 -access r
57
58
59 imodeladdmmregister  -name rx_header -addressblock rxNonSecurePort2/rxRegs
   ↪ -writefunction  txWriteHeader -offset 0x2 -width 2 -access w
60
61 imodeladdmmregister  -name rx_data -addressblock rxNonSecurePort2/rxRegs
   ↪ -writefunction  txWrite -offset 0x4  -access w
```