Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de um Processador RISC-V com Coprocessador Dedicado para Hash Criptográfico

Gustavo Henrique Fernandes Carvalho

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação de um Processador RISC-V com Coprocessador Dedicado para Hash Criptográfico

Gustavo Henrique Fernandes Carvalho

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador)
CIC/UnB

Prof. Dr. Marcus Vinicius Lamar
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 16 de Julho de 2019

Dedicatória

Dedico este trabalho aos meus pais por todo apoio que sempre me deram.

Agradecimentos

Agradeço a minha família por toda motivação para concluir esse curso.
Ao professor Dr. Ricardo Pezzuol Jacobi por ter orientado esse trabalho.
À Universidade de Brasília e a todos os seus professores.
E a todos que direta ou indiretamente fizeram parte da minha formação.

Resumo

Com a crescente preocupação com a segurança de sistemas computacionais, a utilização de *hardware* dedicado e otimizado para o cálculo de funções criptográficas é cada vez mais comum. No desenvolvimento deste trabalho foi implementado um processador com o conjunto de instruções RISC-V e um coprocessador com funções de *hash* criptográfico implementadas em *hardware*. Para garantir o correto funcionamento do processador e do coprocessador foi desenvolvido um conjunto extensivo de testes automatizados. Os resultados mostraram um aumento significativo no custo do *hardware* e um enorme ganho de performance no cálculo de *hashes*.

Palavras-chave: RISC-V, FPGA, hash criptográfico

Abstract

Due to the growing concern with the security of computer systems, the use of dedicated and optimized hardware for compute cryptographic functions is becoming more common. In the development of this work, it was implemented a processor with the RISC-V instruction set architecture and a coprocessor with cryptographic hash functions implemented in hardware. To ensure the correct operation of the processor and coprocessor an extensive set of automated tests has been developed. The results showed a significant increase in the cost of the hardware and a huge performance gain in the computation of hashes.

Keywords: RISC-V, FPGA, cryptographic hash

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CRC *Cyclic redundancy check* - Verificação de redundância cíclica.

FPGA *Field Programmable Gate Array* - Matriz de Portas Programáveis em Campo.

ISA *Instruction Set Architecture* - Arquitetura de Conjunto de Instruções.

MARS *MIPS Assembler and Runtime Simulator* - Simulador de tempo de execução e montador MIPS.

RARS *RISC-V Assembler and Runtime Simulator* - Simulador de tempo de execução e montador RISC-V.

RISC *Reduced Instruction Set Computer* - Computador Com Conjunto Reduzido de Instruções.

SoC *System On Chip* - Sistema em Chip.

Capítulo 1

Introdução

Os movimentos de código aberto, comumente chamados de *open-source*, tiveram grande crescimento e impacto comercial nos últimos anos. Uma aplicação *open-source* pode ser definida como um artefato tangível (*hardware*) ou não tangível (*software*) que qualquer pessoa pode acessar, utilizar, modificar e compartilhar de acordo com a licença sobre a qual o artefato foi distribuído [1, 2].

Hoje temos acesso a uma enorme quantidade de projetos de *software* com código aberto (*open-source software*), desde sistemas de banco de dados relacionais como o MySQL a sistemas operacionais completos como o Parabola GNU/Linux-libre. Os projetos de *software open-source* são responsáveis por grandes inovações tecnológicas e influenciam diretamente grandes aplicações comerciais como o Facebook que foi criado em PHP, o Twitter e Spotify que utilizam Rails e o Uber que utiliza o Node.js, todos projetos *open-source* [3].

Enquanto isso, o movimento de *hardware* aberto (*open-source hardware*) é muito menor em comparação a *software open-source* e as poucas iniciativas que existem são focadas em placas de circuito impresso como o Arduino ou em sistemas parcialmente *open-source*, como a GoPro e o Fitbit [3] (veja Figura 1.1).

Projetos *open-source* de módulos de *hardware* em nível de semicondutores para sistemas em chip (SoC) e para circuitos integrados configuráveis, como FPGA, são escassos [3]. Das poucas iniciativas que existem destacam-se o RISC-V Rocket, OpenRISC, BERI e OpenSPARC que são implementações *open-source* de CPUs, o MIAOW, Nyami e Nyuzi que são implementações *open-source* de GPUs e o OpenCores que é um grande repositório de módulos de *hardware open-source* [3], porém essas iniciativas ainda não são muito utilizadas em aplicações comerciais.

Com o objetivo de desenvolver uma arquitetura de conjunto de instruções (*instruction set architecture*, ISA) totalmente *open-source* e preparada para ser utilizada em aplicações comerciais, os alunos Andrew Waterman e Yunsup Lee da Universidade da Califórnia -

Berkley, orientados pelo professor Dr. David Andrew Patterson começaram a desenvolver uma nova ISA chamada RISC-V [4]. Em 2015 o projeto saiu da Universidade da Califórnia e passou a ser controlado pela RISC-V Foundation, uma organização formada por mais de 250 membros e patrocinadores que incluem grandes empresas como Google, Micron, NVIDIA, Qualcomm e Samsung [5].

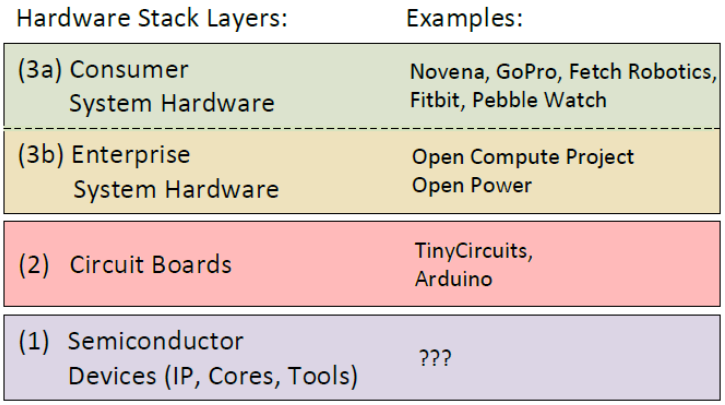

Figura 1.1: Iniciativas de hardware *open-source* (Fonte: [1]).

1.1 Motivação

Um dos principais objetivos do RISC-V é ser uma arquitetura universal, isto é, ser adequada para uso em diversas aplicações diferentes, desde sistemas embarcados de baixa potência a supercomputadores de alta performance [6, 7]. Para atingir esse objetivo a ISA RISC-V foi projetada de forma modular, com extensões opcionais padronizadas e extensões não padronizadas que podem ser adicionadas de acordo com os requisitos da aplicação [8].

Com a crescente preocupação com a segurança de sistemas computacionais, várias ISAs implementam instruções que visam otimizar funções criptográficas, como por exemplo, o x86 utilizado em processadores da Intel e AMD possui a extensão AES-NI [9], a Intel também possui uma extensão proprietária para hash criptográfico, a *Intel® SHA Extensions* [10], e o ARM possui a *ARM® Cortex® Core Cryptographic Extension* [11].

Para o RISC-V conseguir concorrer com as ISAs que dominam o mercado atual de processadores em aplicações que utilizam muitas funções criptográficas, ele irá precisar de otimizações em *hardware* para essas funções.

1.2 Objetivos

O objetivo desse trabalho é implementar um processador RISC-V com o conjunto básico de instruções RV32I, com *pipeline* de 5 estágios e um coprocessador com funções de hash criptográfico implementadas e otimizadas em hardware.

Para garantir o correto funcionamento da implementação, foi desenvolvido um conjunto de testes automatizados.

Para demonstrar o funcionamento do projeto, ele deverá ser sintetizável em uma FPGA e calcular corretamente o *hash* de diferentes dados utilizando os módulos do coprocessador.

Capítulo 2

Referencial Teórico

Esse capítulo apresenta os fundamentos teóricos e os conceitos utilizados no desenvolvimento do projeto. A seção 2.1 introduz a arquitetura RISC-V e os conceitos de arquitetura de computadores, a seção 2.2 explica os conceitos de hash criptográfico, a seção 2.3 detalha o funcionamento dos algoritmos utilizados e a seção 2.4 apresenta as principais ferramentas utilizadas no projeto.

2.1 RISC-V

O RISC-V é uma arquitetura de conjunto de instruções (ISA) *open-source* com um conjunto reduzido de instruções (RISC).

Por ser uma arquitetura recente, a especificação de seu conjunto básico de instruções e suas extensões estão em constante evolução, podendo ocorrer alterações na especificação oficial da ISA que não serão compatíveis com versões anteriores. Para o propósito desse projeto foi utilizado a ISA RISC-V definida no manual do conjunto de instruções versão 2.2 [8].

A ISA RISC-V possui os 4 conjuntos básicos de instruções descritos a seguir:

- RV32I - Conjunto básico de instruções com palavras de 32 bits e operações lógicas e aritméticas de números inteiros.
- RV32E - Uma versão reduzida do RV32I projetada para sistemas embarcados.
- RV64I - Conjunto de instruções similar ao RV32I mas com palavras de tamanho de 64 bits, além de algumas instruções extras para trabalhar com palavras de 32 bits.
- RV128I - É similar ao RV64I com suporte a palavras de 128 bits.

De acordo com o manual do conjunto de instruções, o RV32I e o RV64I não devem sofrer alterações significativas nas próximas versões, o RV32E e o RV128I ainda devem sofrer

grandes alterações [8]. O conjunto de instruções base utilizado para o desenvolvimento desse projeto é o RV32I.

2.1.1 Instruções

Todas as instruções do RV32I possuem tamanho fixo de 32 bits e são divididos em 6 tipos, conforme a Figura 2.1.

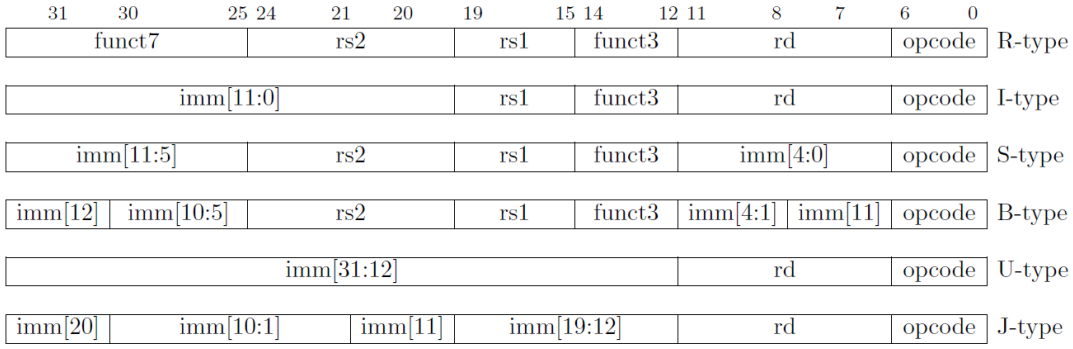

Figura 2.1: Formatos de instruções do RV32I (Fonte: [8]).

Os formatos das instruções do RV32I foram projetados para simplificar a decodificação, mantendo os valores de rs1, rs2 e rd na mesma posição em todos os tipos de instruções e nas instruções que possuem valor imediato, o bit 31 da instrução sempre será igual ao bit 31 do imediato, pois em todas as instruções o sinal do imediato é estendido [12].

A ISA RISC-V também possui suporte a instruções com tamanhos variados, porém eles são utilizados apenas por algumas extensões e não foram utilizados nesse projeto.

2.1.2 Extensões

O RISC-V foi projetado para ter suporte a extensões oficiais e não oficiais. As extensões oficiais da ISA são identificadas por apenas uma letra. Algumas das extensões definidas na versão 2.2 do manual [8] da ISA são as seguintes:

- M - Multiplicação e divisão de inteiros.
- A - Operações atômicas.
- F - Operações de ponto flutuante com precisão simples compatível com o padrão IEEE 754-2008 e adiciona 32 registradores de 32 bits para ponto flutuante.
- D - Operações de ponto flutuante com precisão dupla, requer a extensão F. Os registradores de ponto flutuante são estendidos para 64 bits.

- Q - Operações de ponto flutuante com precisão quádrupla, requer as extensões F e D. Os registradores de ponto flutuante são estendidos para 128 bits.
- C - Adiciona instruções de 16 bits para reduzir o tamanho do código.
- N - Adiciona instruções para tratamento de exceções e interrupções em nível de usuário.

O conjunto das instruções RV32IMAFD é chamado apenas de RV32G.

Além de oferecer extensões oficiais, outro objetivo do RISC-V é servir como base para o desenvolvimento de conjuntos de instruções especializados e aceleradores customizados [8]. Para isso a ISA deixa vários *opcodes* disponíveis para instruções customizadas, além de permitir que as instruções customizadas possuam tamanhos diferentes.

Como pode ser observado na Figura 2.2, para instruções de 32 bits existem 4 *opcodes* marcados como *custom* e que podem ser utilizados por extensões não oficiais do conjunto de instruções. Os *opcodes custom-0* e *custom-1* são reservados apenas para instruções customizadas e o *custom-2* e o *custom-3* compartilham o mesmo *opcode* de instruções para operações em 128 bits, mas podem ser utilizados para instruções customizadas quando a implementação não terá suporte a 128 bits.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

Figura 2.2: Mapa de *opcodes* do RISC-V (Fonte: [8]).

2.1.3 Arquitetura *pipeline*

A técnica de *pipeline* consiste em dividir as instruções em etapas, que utilizam apenas uma fatia dos recursos disponíveis, permitindo que várias etapas de diferentes instruções possam ser executadas ao mesmo tempo [13].

A Figura 2.3 ilustra a execução de instruções em uma arquitetura *single-cycle* e uma *pipeline*, nela podemos observar que a arquitetura *pipeline* não diminui o tempo de latência, isto é, o tempo para uma instrução ser executada.

O ganho do *pipeline* é o número de instruções executadas por um período de tempo (*throughput*) [13]. Idealmente uma arquitetura *pipeline* de n estágios iria executar n vezes mais instruções que uma arquitetura *single-cycle*, porém em aplicações reais o ganho não chega a ser n devido a latência, riscos de dados e riscos de controle.

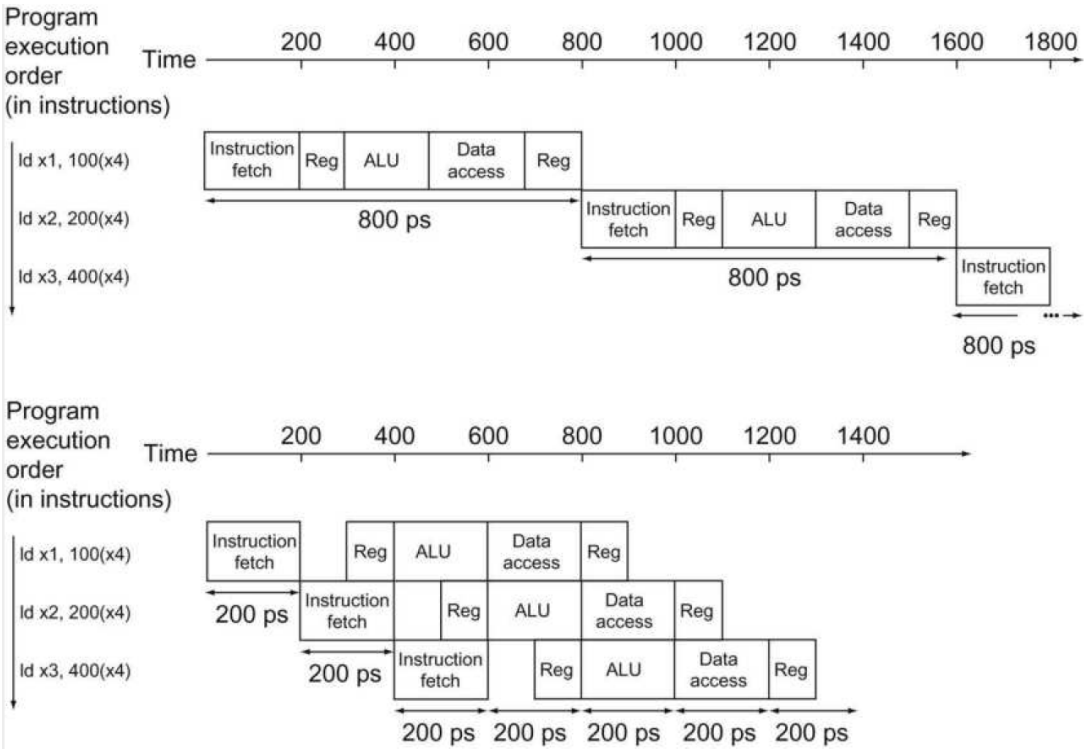

Figura 2.3: Execução *single-cycle* vs *pipeline* (Fonte: [13]).

Mesmo com a complexidade extra de uma arquitetura *pipeline* e os riscos de controle e dados que devem ser tratados, na maioria das aplicações ela ainda apresenta um desempenho melhor que as arquiteturas *single-cycle* e *multi-cycle*. Hoje em dia a arquitetura *pipeline* é quase universal para arquitetura de computadores [13].

Pipeline hazards

Risco de dados, ou *data hazard*, é a situação em que uma instrução não pode ser executada porque ela depende do resultado de outra instrução que ainda está sendo executada no pipeline. A solução para esse problema é manter a instrução no estado em que o *data hazard* ocorreu e bloquear os estados anteriores para que não avancem no *pipeline*, além disso são adicionadas instruções que não possuem efeito colateral, chamadas bolhas, no estado seguinte ao *data hazard* para que as instruções a frente da instrução que causou o *data hazard* possam seguir a execução no *pipeline* e serem concluídas, assim resolvendo o *data hazard*.

Em um pipeline sem *hazards*, após o tempo de latência da primeira instrução, uma nova instrução será concluída a cada ciclo de clock, porém quando são adicionadas bolhas para resolver *data hazards*, terão ciclos de clock que nenhuma instrução será concluída, assim reduzindo o *throughput*.

Uma técnica para reduzir o número de bolhas necessárias para resolver um *data hazard* é o adiantamento de dados, ou *forwarding*. O adiantamento de dados consiste em identificar os pontos em que podem ocorrer *data hazard* e criar meios de recuperar os dados necessários dos *buffers* internos ao invés de esperar o dado ser escrito nos registradores [13].

Outro tipo de risco do *pipeline* é o risco de controle, ou *control hazard*. Ele ocorre quando a decisão de realizar ou não um pulo condicional ocorre após outras instruções já terem sido carregadas nos estágios anteriores do *pipeline*. Quando isso acontece é necessário descartar as instruções carregadas após a instrução de pulo.

Pipeline do RISC-V

Existem várias maneiras de separar as tarefas de um processador em etapas de uma arquitetura *pipeline*. A arquitetura utilizada nesse projeto foi baseada no livro *Computer Organization and Design RISC-V Edition: The Hardware Software Interface* [13] e consiste em um *pipeline* com as 5 etapas descritas a seguir:

1. *Instruction Fetch (IF)* - Nessa etapa é realizada a leitura da memória de instruções e a atualização do contador de programa (PC).
2. *Instruction Decode (ID)* - Nessa etapa a instrução é decodificada e de acordo com a instrução são realizadas as leituras no banco de registradores e/ou extensão de sinal do imediato.
3. *Execution (EX)* - Nessa etapa são realizadas as operações lógicas e aritméticas.
4. *Data Memory Access (MEM)* - Nessa etapa é realizada a escrita ou a leitura da memória de dados.
5. *Write Back (WB)* - Nessa etapa é realizada a escrita no banco de registradores.

2.2 Hash criptográfico

A Figura 2.4 ilustra a definição de uma função de *hash*, isto é, uma função que recebe uma sequencia de bits qualquer, normalmente chamada de *message* ou M, e gera uma saída de tamanho fixo chamada de *message digest* ou H.

Existem vários tipos de funções de hash como, por exemplo, o *Cyclic Redundancy Check* (CRC) comumente utilizado para detectar erros no armazenamento e transmissão de dados, o *universal hashing functions* utilizados em tabelas de *hash*, o *keyed hash* utilizado para autenticação, além de vários outros tipos e aplicações de *hash*.

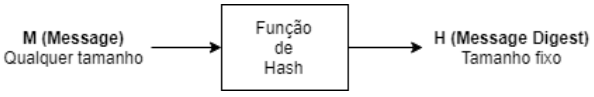

Figura 2.4: Definição de uma função de hash.

Uma das categorias de funções de *hash* é o *hash* criptográfico, porém existem críticas a essa categorização de algoritmos de *hash* por causa da falta de uma definição formal das propriedades que um algoritmo deve ter para ser considerado de *hash* criptográfico. O livro *Handbook of Applied Cryptography* [14] define que um algoritmo de *hash* criptográfico deve possuir as seguintes propriedades:

1. *preimage resistance* - Deve ser computacionalmente inviável encontrar uma entrada que gere uma determinada saída, isto é, dado uma saída y de uma função de *hash* $h()$, deve ser inviável encontrar um x que satisfaça $h(x) = y$.
2. *2^{nd}-preimage resistance* - Deve ser computacionalmente inviável encontrar uma segunda entrada que gere a mesma saída que uma outra entrada específica, isto é, dado uma entrada x, deve ser inviável encontrar uma entrada x' tal que $x \neq x'$ e $h(x) = h(x')$.
3. *collision resistance* - Deve ser computacionalmente inviável encontrar duas entradas qualquer que gerem a mesma saída, isto é, para qualquer x, deve ser inviável encontrar uma entrada x' tal que $x \neq x'$ e $h(x) = h(x')$.

A principal crítica à definição anterior de *hash* criptográfico é que a interpretação de um processamento computacional inviável é dependente da tecnologia disponível, ou seja, um algoritmo de *hash* considerado seguro para uso criptográfico hoje pode deixar de ser seguro com o avanço da tecnologia, como foi o caso do MD5.

O algoritmo MD5 foi definido como sendo uma função de *hash* para uso criptográfico pela RFC 1321 de 1992 [15] e após várias demostrações e pesquisas mostrarem que o MD5 não satisfaz mais as propriedades de um algoritmo de *hash* criptográfico, em 2011 foi publicada a RFC 6151 que atualizou oficialmente as considerações de segurança do MD5 e desaconselhou sua utilização em aplicações de segurança [16]. Por ter sido o algoritmo de *hash* mais utilizado por tanto tempo e várias aplicações legadas dependerem dele, o MD5 ainda é muito utilizado mesmo com suas falhas de segurança.

Os algoritmos de *hash* são extremamente versáteis e são utilizados por inúmeros algoritmos criptográficos e protocolos de segurança como assinaturas digitais, criptografia de chave pública, verificação de integridade, autenticação de mensagens, proteção de senhas, protocolos de acordo chave e muitos outros [17].

Tabela 2.1: Tamanho da entrada e do *message digest* de diferentes algoritmos de hash.

Algoritmo	Tamanho do bloco de entrada (bits)	Tamanho do *message digest* (bits)
MD5	512	128
SHA-1	512	160
SHA-256	512	256
SHA-512	1024	512

Os algoritmos implementados nesse projeto foram o MD5, SHA-1, SHA-256 e SHA-512. A tabela Tabela 2.1 mostra a diferença do tamanho dos blocos de entrada e de saída (*message digest*) de cada um desses algoritmos. A seção 2.3 apresenta em mais detalhes o funcionamento desses algoritmos.

2.3 Definição dos algoritmos de hash criptográficos utilizados

Nessa seção será apresentada a ideia geral do funcionamento dos algoritmos MD5, SHA-1, SHA-256 e SHA-512. Informações mais detalhadas sobre o funcionamento dos algoritmos podem ser encontradas em suas respectivas RFCs [15, 18, 19].

As seguintes definições e terminologias serão utilizadas na descrição dos algoritmos:

- Nos algoritmos MD5, SHA-1 e SHA-256 uma palavra corresponde a 32 bits e no SHA-512 uma palavra corresponde a 64 bits.
- Dado que uma mensagem de um tamanho qualquer seja dividida em blocos de tamanho iguais, então X_j representa o j-ésimo bloco da mensagem e $X_j[k]$ é a k-ésima palavra do j-ésimo bloco.
- O resultado do operador *mod* é o resto da divisão inteira dos dois operandos.
- $(x >> n)$ e $(x << n)$ representam a operação de deslocamento lógico dos bits de x, n vezes para a direita ou para a esquerda, respectivamente.
- $(x >>> n)$ e $(x <<< n)$ representam a operação de deslocamento circular lógico dos bits de x, n vezes para a direita ou para a esquerda, respectivamente.

2.3.1 MD5

De acordo com a especificação do MD5 na RFC 6151 [15], o algoritmo consiste nos 4 passos descritos a seguir.

Etapa de preenchimento e análise da mensagem

Nessa etapa são adicionados bits à mensagem de forma que o número de bits da mensagem seja múltiplo de 512. Considerando uma mensagem de tamanho original L, os bits são adicionados conforme os passos a seguir:

1. Um bit '1' é adicionado à mensagem.
2. K bits '0' são adicionados à mensagem, onde K é o menor inteiro não negativo que satisfaz a equação $(L + 1 + K)\ mod\ 512 = 448$. Após esse passo, o resto da divisão inteira (módulo) do tamanho da mensagem por 512 será 448.
3. Nesse passo são adicionados 64 bits à mensagem, o valor desses bits representa o tamanho original da mensagem L. Esses 64 bits são adicionados como duas palavras de 32 bits em *low-order*, isto é, a palavra menos significativa é adicionada primeiro. Após esse passo o tamanho da mensagem será múltiplo de 512.

Inicialização do *buffer*

O *message digest* do MD5 é armazenado em um *buffer* de 4 palavras de 32 bits, sendo elas H0, H1, H2 e H3. O *buffer* é inicializado de acordo com a Equação 2.1.

$$\begin{aligned} H0 &= \text{0x01234567} \\ H1 &= \text{0x89}ABCDEF \\ H2 &= \text{0x}FEDCBA\text{98} \\ H3 &= \text{0x76543210} \end{aligned} \tag{2.1}$$

Processamento

É nessa etapa que o *message digest* é calculado. Primeiramente a mensagem é dividida em blocos de 512 bits e o processamento de cada bloco é dividido em 64 passos.

Para cada bloco o valor atual do *message digest* é copiado para um *buffer* que será modificado durante os passos do processamento do bloco, conforme a Equação 2.2.

$$\begin{aligned} A &= H0 \\ B &= H1 \\ C &= H2 \\ D &= H3 \end{aligned} \tag{2.2}$$

Após copiar o *buffer*, são realizados os 64 passos do algoritmo sobre o bloco, onde cada passo é representado por i que varia de 0 a 63. Em cada passo são realizadas as operações da Equação 2.3.

$$
\begin{aligned}
T &= B + ((A + Func(B, C, D) + X_j[k] + T[i]) <<< s[i]) \\
A &= D \\
B &= T \\
C &= B \\
D &= C
\end{aligned}
\tag{2.3}
$$

Onde X_j representa o bloco que está sendo processado e $X_j[k]$ é a k-ésima palavra do bloco. T e s são tabelas de constantes definidas na RFC 6151 [15], $T[i]$ e $s[i]$ são o i-ésimo elemento dessas tabelas. A função $Func(B, C, D)$ é definida para cada passo de acordo com a Equação 2.4.

$$
Func(B, C, D) = \begin{cases}
(B \wedge C) \vee (\neg B \wedge D), & 0 \leq i < 16 \\
(B \wedge D) \vee (C \wedge \neg D), & 16 \leq i < 32 \\
B \oplus C \oplus D, & 32 \leq i < 48 \\
C \oplus (B \oplus \neg D), & 48 \leq i < 63
\end{cases}
\tag{2.4}
$$

Ao final do processamento do bloco, o *buffer* do *message digest* é atualizado de acordo com a Equação 2.5. Se o bloco atual for o último bloco da mensagem a etapa de processamento acabou, caso contrário o j é incrementado e o processamento se repete para o próximo bloco.

$$
\begin{aligned}
H0 &= H0 + A \\
H1 &= H1 + B \\
H2 &= H2 + C \\
H3 &= H3 + D
\end{aligned}
\tag{2.5}
$$

Saída

Ao final do processamento o *message digest* estará no *buffer* H0, H1, H2 e H3, onde o H0 é a palavra mais significativa e o H3 é a menos significativa e os bytes dentro de cada palavra estão em *low-order*, isto é, o primeiro byte de cada palavra é o menos significativo e o último é o mais significativo.

2.3.2 SHA-1

De acordo com a especificação do SHA-1 na RFC 3174 [18], o algoritmo consiste nos 4 passos descritos a seguir.

Etapa de preenchimento e análise da mensagem

Nessa etapa são adicionados bits à mensagem de forma que o número de bits da mensagem seja múltiplo de 512. Considerando uma mensagem de tamanho original L, os bits são adicionados conforme os passos a seguir:

1. Um bit '1' é adicionado à mensagem.
2. K bits '0' são adicionados à mensagem, onde K é o menor inteiro não negativo que satisfaz a equação $(L + 1 + K)\ mod\ 512 = 448$. Após esse passo, o resto da divisão inteira (módulo) do tamanho da mensagem por 512 será 448.
3. Nesse passo são adicionados 64 bits à mensagem, o valor desses bits representa o tamanho original da mensagem L.

Inicialização do *buffer*

O *message digest* do SHA-1 é armazenado em um *buffer* de 5 palavras de 32 bits, sendo elas H0, H1, H2, H3 e H4. O *buffer* é inicializado de acordo com a Equação 2.6.

$$\begin{aligned} H0 &= \text{0x67452301} \\ H1 &= \text{0x}EFCDAB89 \\ H2 &= \text{0x}98BADCFE \\ H3 &= \text{0x10325476} \\ H4 &= \text{0x}C3D2E1F0 \end{aligned} \tag{2.6}$$

Processamento

É nessa etapa que o *message digest* é calculado. Primeiramente a mensagem é dividida em blocos de 512 bits e o processamento de cada bloco é dividido em 80 passos.

Para cada bloco o valor atual do *message digest* é copiado para um *buffer* que será modificado durante os passos do processamento do bloco, conforme a Equação 2.7.

$$\begin{aligned} A &= H0 \\ B &= H1 \\ C &= H2 \\ D &= H3 \\ E &= H4 \end{aligned} \tag{2.7}$$

Após copiar o *buffer*, o bloco atual é dividido em 16 palavras $W(0), W(1), ..., W(15)$, onde o $W(0)$ é a palavra mais a esquerda. O *buffer* W é formado por 80 palavras, sendo

que as 16 primeiras são preenchidas com o bloco da mensagem e outras 64 são preenchidas com a Equação 2.8.

$$\begin{gathered}\text{Para } 16 \leq t \leq 79: \\ W(t) = (W(t-3) \oplus W(t-8) \oplus W(t-14) \oplus W(t-16)) <<< 1\end{gathered} \tag{2.8}$$

O processamento do bloco é realizado em 80 passos, o passo atual é representado por t que varia de 0 a 79. Em cada passo são realizadas as operações da Equação 2.9.

$$\begin{aligned} T1 &= (A <<< 5) + F(t, B, C, D) + E + W(t) + K(t) \\ T2 &= B <<< 30 \\ A &= T1 \\ B &= A \\ C &= T2 \\ D &= C \\ E &= D \end{aligned} \tag{2.9}$$

Onde a função $F(t, B, C, D)$ é definida para cada passo de acordo com a Equação 2.10 e o valor de $K(t)$ é dado pela Equação 2.11.

$$F(t, B, C, D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D), & 0 \leq t \leq 19 \\ B \oplus C \oplus D, & 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D), & 40 \leq t \leq 59 \\ B \oplus C \oplus D, & 60 \leq t \leq 79 \end{cases} \tag{2.10}$$

$$K(t) = \begin{cases} \text{0x}5A827999, & 0 \leq t \leq 19 \\ \text{0x}6ED9EBA1, & 20 \leq t \leq 39 \\ \text{0x}8F1BBCDC, & 40 \leq t \leq 59 \\ \text{0x}CA62C1D6, & 60 \leq t \leq 79 \end{cases} \tag{2.11}$$

Ao final do processamento do bloco, o *buffer* do *message digest* é atualizado de acordo com a Equação 2.12. Se o bloco atual for o último bloco da mensagem a etapa de processamento acabou, caso contrário o processamento se repete para o próximo bloco.

$$
\begin{aligned}
H0 &= H0 + A \\
H1 &= H1 + B \\
H2 &= H2 + C \\
H3 &= H3 + D \\
H4 &= H4 + E
\end{aligned}
\tag{2.12}
$$

Saída

Ao final do processamento o *message digest* estará no *buffer* H0, H1, H2, H3 e H4, onde o H0 é a palavra mais significativa e o H4 é a menos significativa.

2.3.3 SHA-256

De acordo com a especificação do SHA-256 na RFC 4634 [19], o algoritmo consiste nos 4 passos descritos a seguir.

Etapa de preenchimento e análise da mensagem

Nessa etapa são adicionados bits à mensagem de forma que o número de bits da mensagem seja múltiplo de 512. Considerando uma mensagem de tamanho original L, os bits são adicionados conforme os passos a seguir:

1. Um bit '1' é adicionado à mensagem.
2. K bits '0' são adicionados à mensagem, onde K é o menor inteiro não negativo que satisfaz a equação $(L + 1 + K)\ mod\ 512 = 448$. Após esse passo, o resto da divisão inteira (módulo) do tamanho da mensagem por 512 será 448.
3. Nesse passo são adicionados 64 bits à mensagem, o valor desses bits representa o tamanho original da mensagem L.

Inicialização do *buffer*

O *message digest* do SHA-256 é armazenado em um *buffer* de 8 palavras de 32 bits, sendo elas H0, H1, H2, H3, H4, H5, H6 e H7. O *buffer* é inicializado de acordo com a Equação 2.13.

$$
\begin{aligned}
H0 &= 0\text{x}6A09E667 \\
H1 &= 0\text{x}BB67AE85 \\
H2 &= 0\text{x}3C6EF372 \\
H3 &= 0\text{x}A54FF53A \\
H4 &= 0\text{x}510E527F \\
H5 &= 0\text{x}9B05688C \\
H6 &= 0\text{x}1F83D9AB \\
H7 &= 0\text{x}5BE0CD19
\end{aligned}
\tag{2.13}
$$

Processamento

É nessa etapa que o *message digest* é calculado. Primeiramente a mensagem é dividida em blocos de 512 bits e o processamento de cada bloco é dividido em 64 passos.

Para cada bloco o valor atual do *message digest* é copiado para um *buffer* que será modificado durante os passos do processamento do bloco, conforme a Equação 2.14.

$$
\begin{aligned}
A &= H0 \\
B &= H1 \\
C &= H2 \\
D &= H3 \\
E &= H4 \\
F &= H5 \\
G &= H6 \\
H &= H7
\end{aligned}
\tag{2.14}
$$

Após copiar o *buffer*, o bloco atual é dividido em 16 palavras $W(0), W(1), ..., W(15)$, onde o $W(0)$ é a palavra mais a esquerda. O *buffer* W é formado por 64 palavras, sendo que as 16 primeiras são preenchidas com o bloco da mensagem e outras 48 são preenchidas com a Equação 2.15.

$$
\begin{aligned}
&\text{Para } 16 \leq t \leq 63: \\
W(t) &= SSIG1(W(t-2)) + W(t-7) + SSIG0(t-15) + W(t-16) \\
SSIG0(x) &= (x >>> 7) \oplus (x >>> 18) \oplus (x >> 3) \\
SSIG1(x) &= (x >>> 17) \oplus (x >>> 19) \oplus (x >> 10)
\end{aligned}
\tag{2.15}
$$

O processamento do bloco é realizado em 64 passos, o passo atual é representado por t que varia de 0 a 63. Em cada passo são realizadas as operações da Equação 2.16.

$$\begin{aligned} T1 &= H + BSIG1(E) + CH(E,F,G) + K(t) + W(t) \\ T2 &= BSIG0(A) + MAJ(A,B,C) \\ A &= T1 + T2 \\ B &= A \\ C &= B \\ D &= C \\ E &= D + T1 \\ F &= E \\ G &= F \\ H &= G \end{aligned} \tag{2.16}$$

Onde K é uma tabela de constantes definida na RFC 4634 [19] e $K(t)$ é o t-ésimo valor dessa tabela e as funções $CH(x,y,z)$, $MAJ(x,y,z)$, $BSIG0(x)$ e $BSIG1(x)$ são definidas na Equação 2.17.

$$\begin{aligned} CH(x,y,z) &= (x \wedge y) \oplus ((\neg x) \wedge z) \\ MAJ(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ BSIG0(x) &= (x >>> 2) \oplus (x >>> 13) \oplus (x >>> 22) \\ BSIG1(x) &= (x >>> 6) \oplus (x >>> 11) \oplus (x >>> 25) \end{aligned} \tag{2.17}$$

Ao final do processamento do bloco, o *buffer* do *message digest* é atualizado de acordo com a Equação 2.18. Se o bloco atual for o último bloco da mensagem a etapa de processamento acabou, caso contrário o processamento se repete para o próximo bloco.

$$\begin{aligned} H0 &= H0 + A \\ H1 &= H1 + B \\ H2 &= H2 + C \\ H3 &= H3 + D \\ H4 &= H4 + E \\ H5 &= H5 + F \\ H6 &= H6 + G \\ H7 &= H7 + H \end{aligned} \tag{2.18}$$

Saída

Ao final do processamento o *message digest* estará no *buffer* H0, H1, H2, H3, H4, H5, H6 e H7, onde o H0 é a palavra mais significativa e o H7 é a menos significativa.

2.3.4 SHA-512

A principal diferença do SHA-512 em relação aos algoritmos analisados anteriormente é que o tamanho de sua palavra é 64 bits.

De acordo com a especificação do SHA-512 na RFC 4634 [19], o algoritmo consiste nos 4 passos descritos a seguir.

Etapa de preenchimento e análise da mensagem

Nessa etapa são adicionados bits à mensagem de forma que o número de bits da mensagem seja múltiplo de 1024. Considerando uma mensagem de tamanho original L, os bits são adicionados conforme os passos a seguir:

1. Um bit '1' é adicionado à mensagem.
2. K bits '0' são adicionados à mensagem, onde K é o menor inteiro não negativo que satisfaz a equação $(L + 1 + K)\ mod\ 1024 = 896$. Após esse passo, o resto da divisão inteira (módulo) do tamanho da mensagem por 1024 será 896.
3. Nesse passo são adicionados 128 bits à mensagem, o valor desses bits representa o tamanho original da mensagem L.

Inicialização do *buffer*

O *message digest* do SHA-512 é armazenado em um *buffer* de 8 palavras de 64 bits, sendo elas H0, H1, H2, H3, H4, H5, H6 e H7. O *buffer* é inicializado de acordo com a Equação 2.19.

$$\begin{aligned} H0 &= 0\text{x}6A09E667F3BCC908 \\ H1 &= 0\text{x}BB67AE8584CAA73B \\ H2 &= 0\text{x}3C6EF372FE94F82B \\ H3 &= 0\text{x}A54FF53A5F1D36F1 \\ H4 &= 0\text{x}510E527FADE682D1 \\ H5 &= 0\text{x}9B05688C2B3E6C1F \\ H6 &= 0\text{x}1F83D9ABFB41BD6B \\ H7 &= 0\text{x}5BE0CD19137E2179 \end{aligned} \tag{2.19}$$

Processamento

É nessa etapa que o *message digest* é calculado. Primeiramente a mensagem é dividida em blocos de 1024 bits e o processamento de cada bloco é dividido em 80 passos.

Para cada bloco o valor atual do *message digest* é copiado para um *buffer* que será modificado durante os passos do processamento do bloco, conforme a Equação 2.20.

$$\begin{aligned} A &= H0 \\ B &= H1 \\ C &= H2 \\ D &= H3 \\ E &= H4 \\ F &= H5 \\ G &= H6 \\ H &= H7 \end{aligned} \tag{2.20}$$

Após copiar o *buffer*, o bloco atual é dividido em 16 palavras $W(0), W(1), ..., W(15)$, onde o $W(0)$ é a palavra mais a esquerda. O *buffer* W é formado por 80 palavras, sendo que as 16 primeiras são preenchidas com o bloco da mensagem e outras 64 são preenchidas de acordo com a Equação 2.21.

$$\begin{gathered} \text{Para } 16 \leq t \leq 79: \\ W(t) = SSIG1(W(t-2)) + W(t-7) + SSIG0(t-15) + W(t-16) \\ SSIG0(x) = (x >>> 1) \oplus (x >>> 8) \oplus (x >> 7) \\ SSIG1(x) = (x >>> 19) \oplus (x >>> 61) \oplus (x >> 6) \end{gathered} \tag{2.21}$$

O processamento do bloco é realizado em 80 passos, o passo atual é representado por t que varia de 0 a 79. Em cada passo são realizadas as operações da Equação 2.22.

$$
\begin{aligned}
T1 &= H + BSIG1(E) + CH(E,F,G) + K(t) + W(t) \\
T2 &= BSIG0(A) + MAJ(A,B,C) \\
A &= T1 + T2 \\
B &= A \\
C &= B \\
D &= C \\
E &= D + T1 \\
F &= E \\
G &= F \\
H &= G
\end{aligned}
\tag{2.22}
$$

Onde K é uma tabela de constantes definida na RFC 4634 [19] e $K(t)$ é o t-ésimo valor dessa tabela e as funções $CH(x,y,z)$, $MAJ(x,y,z)$, $BSIG0(x)$ e $BSIG1(x)$ são definidas na Equação 2.23.

$$
\begin{aligned}
CH(x,y,z) &= (x \wedge y) \oplus ((\neg x) \wedge z) \\
MAJ(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
BSIG0(x) &= (x >>> 28) \oplus (x >>> 34) \oplus (x >>> 39) \\
BSIG1(x) &= (x >>> 14) \oplus (x >>> 18) \oplus (x >>> 41)
\end{aligned}
\tag{2.23}
$$

Ao final do processamento do bloco, o *buffer* do *message digest* é atualizado de acordo com a Equação 2.24. Se o bloco atual for o último bloco da mensagem a etapa de processamento acabou, caso contrário o processamento se repete para o próximo bloco.

$$
\begin{aligned}
H0 &= H0 + A \\
H1 &= H1 + B \\
H2 &= H2 + C \\
H3 &= H3 + D \\
H4 &= H4 + E \\
H5 &= H5 + F \\
H6 &= H6 + G \\
H7 &= H7 + H
\end{aligned}
\tag{2.24}
$$

Saída

Ao final do processamento o *message digest* estará no *buffer* H0, H1, H2, H3, H4, H5, H6 e H7, onde o H0 é a palavra mais significativa e o H7 é a menos significativa.

2.4 Ferramentas utilizadas

Nessa seção são apresentadas as principais ferramentas utilizadas no desenvolvimento do projeto.

2.4.1 VUnit

O VUnit é uma *framework* de código aberto desenvolvida em Python para auxiliar no desenvolvimento de testes automatizados de código VHDL ou Verilog. Essa *framework* permite aplicar técnicas muito utilizadas no desenvolvimento de *softwares* em projetos de *hardware*.

A grande vantagem da utilização de uma *framework* como o VUnit é que o processo de desenvolvimento, testes e correção de erros fica muito mais rápido.

Além de automatizar a execução dos testes, o VUnit também possui um compilador inteligente que compila apenas os arquivos que sofreram alterações desde a última compilação, reduzindo muito o tempo de recompilação em projetos muito grandes.

2.4.2 RARS

O RARS é um simulador e montador da linguagem *assembly* RISC-V que foi desenvolvido sobre o MARS [20].

O MARS é um simulador e montador da linguagem *assembly* MIPS desenvolvido em Java e *open-source* [21]. A organização dos seus módulos e ferramentas torna simples e possível realizar modificações em seu código e possibilitou o desenvolvimento do RARS.

A facilidade de modificar o RARS foi de grande importância para o desenvolvimento desse projeto, pois permitiu a adição de novas funcionalidade que serão discutidas na Seção 3.3.

2.4.3 Quartus Prime

O Quartus Prime é um ambiente para análise e síntese de projetos em FPGAs desenvolvido pela Intel/Altera. Ele inclui várias ferramentas úteis para o desenvolvimento, como por exemplo, o ModelSim para simulação, o Timing Analyser para análise temporal

e o Signal Tap Logic Analyser para ver sinais internos de um projeto sintetizado em uma FPGA.

Além disso o Quartus Prime também faz a síntese do código em alguma linguagem de descrição de *hardware* e programa a FPGA.

2.4.4 RISC-V GNU Toolchain

O RISC-V GNU Toolchain é um conjunto de ferramentas para auxiliar o desenvolvimento de aplicações para a arquitetura RISC-V.

Dentre as ferramentas disponíveis nesse pacote está uma versão do compilador gcc que suporta *cross-copile* de códigos C e C++ para RISC-V, o DejaGnu que é uma *framework* para teste automatizado, o gdb que é *debugger*, o glibc que é uma implementação da biblioteca padrão da linguagem C e o qemu que é um virtualizador que suporta várias arquiteturas, incluindo o RISC-V.

Capítulo 3

Implementação

Esse capítulo apresenta detalhes da implementação do processador RISC-V na seção 3.1, a implementação do coprocessador criptográfico na seção 3.2 e as modificações e ferramentas adicionadas ao RARS na seção 3.3.

Os códigos do processador RISC-V e do coprocessador estão disponíveis em `www.github.com/gustavohfc/RISC-V-Cryptographic-Coprocessor` e as modificações realizadas no RARS estão disponíveis em `www.github.com/gustavohfc/rars`.

3.1 RISC-V

A Figura 3.1 apresenta uma visão *top-level* da implementação do processador RISC-V. Nela podemos observar os módulos dos 5 estágios do *pipeline*, além de um multiplexador e algumas portas para comunicação com módulos externos.

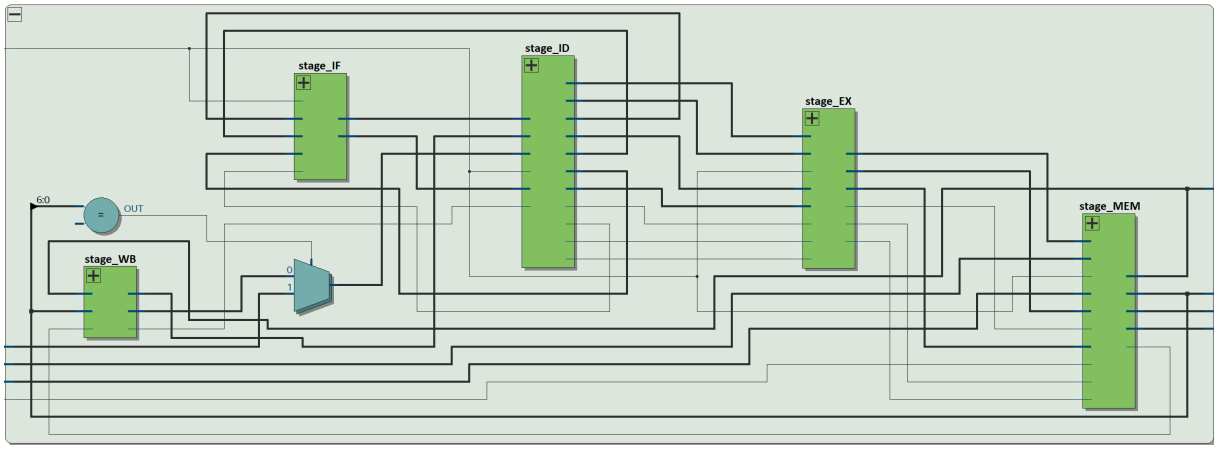

Figura 3.1: Visão *top-level* da implementação do processador RISC-V.

O multiplexador que pode ser visto na Figura 3.1 é utilizado para selecionar se o dado que será escrito no banco de registradores é o dado saindo do estágio WB ou o dado vindo do coprocessador externo, de acordo com o *opcode* da instrução no último estágio do *pipeline*.

As portas que podem ser vistas na Figura 3.1 são utilizadas para comunicação com o coprocessador criptográfico, que será discutido na seção 3.2, e também para permitir que módulos externos tenham acesso direto à memória de dados, utilizado pelos *testbenchs* que serão discutidos no capítulo 4.

A funcionalidade de cada um dos módulos na Figura 3.1 será detalhada nas subseções a seguir.

3.1.1 Módulo stage_IF

O módulo stage_IF contém o registrador do contador de programa (*program counter*, PC) e a memória de instruções. Ele é responsável por ler a próxima instrução de acordo com o PC e passar para o próximo estado.

Além disso, ele também é responsável por atualizar corretamente o valor PC de acordo com a instrução que foi decodificada no estágio ID, podendo ser necessário realizar um salto no código e adicionar uma bolha no *pipeline* ou simplesmente incrementar 4 no valor de PC.

3.1.2 Módulo stage_ID

O módulo stage_ID é responsável por decodificar as instruções e repassar aos outros módulos várias informações sobre como a instrução deverá ser executada, como selecionar as entradas da unidade lógica e aritmética (ULA) do estado EX, indicar ao estado MEM se deve ocorrer escrita na memória ou se o estado WB deve escrever nos registradores.

Esse módulo também é responsável por identificar instruções de pulo, decidir se o pulo deverá ocorrer ou não e repassar ao módulo stage_IF qual deverá ser o próximo PC.

O banco de registradores fica dentro desse módulo e de acordo com a instrução decodificada os valores dos registradores são lidos e repassadas para o módulo stage_EX. A decodificação do imediato, quando existe, também fica dentro desse módulo e o imediato decodificado é passado para os próximos módulos.

Além disso, esse módulo também é responsável por identificar *data hazards* no *pipeline* e inserir bolhas conforme necessário.

3.1.3 Módulo stage_EX

O módulo stage_EX é responsável por realizar as operações lógicas e aritméticas.

3.1.4 Módulo stage_MEM

O módulo stage_MEM é responsável por realizar as operações de leitura e escrita na memória de dados, tratando os casos em que a leitura ou escrita é de apenas um byte, meia palavra ou uma palavra.

Este módulo disponibiliza portas que permitem o acesso direto à memória de dados por módulos externos ao processador RISC-V.

3.1.5 Módulo stage_WB

O módulo stage_WB é responsável por escrever no banco de registradores. A escrita ocorre na borda de decida do *clock* para reduzir em um ciclo o tempo gasto para o valor de um registrador ser atualizado com o resultado de uma operação, assim reduzindo o número de bolhas necessárias para tratar *data hazards*.

3.2 Coprocessador criptográfico

No diagrama da Figura 3.2 podemos observar como o coprocessador desenvolvido se conecta ao pipeline do RISC-V.

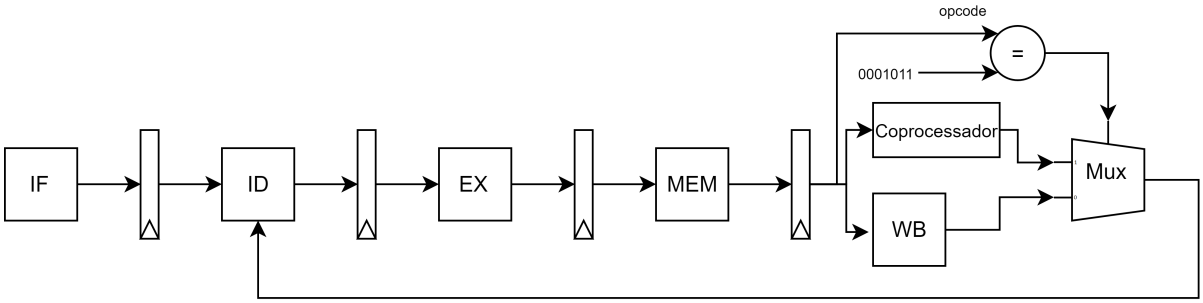

Figura 3.2: Coprocessador no pipeline do RISC-V.

Como pode ser visto na figura Figura 3.3, a comunicação entre o processador principal e o coprocessador é realizada pelos 4 barramentos de 32 bits descritos a seguir:

- coprocessor_instruction - É a instrução que está na última etapa do *pipeline*, ela é decodificada e caso seja uma instrução com *opcode* do coprocessador, sua ação é executada pelo coprocessador.
- coprocessor_data - É o dado lido da memória de dados.
- coprocessor_r2 - É o valor lido do registrador especificado pelo campo rs2 da instrução.

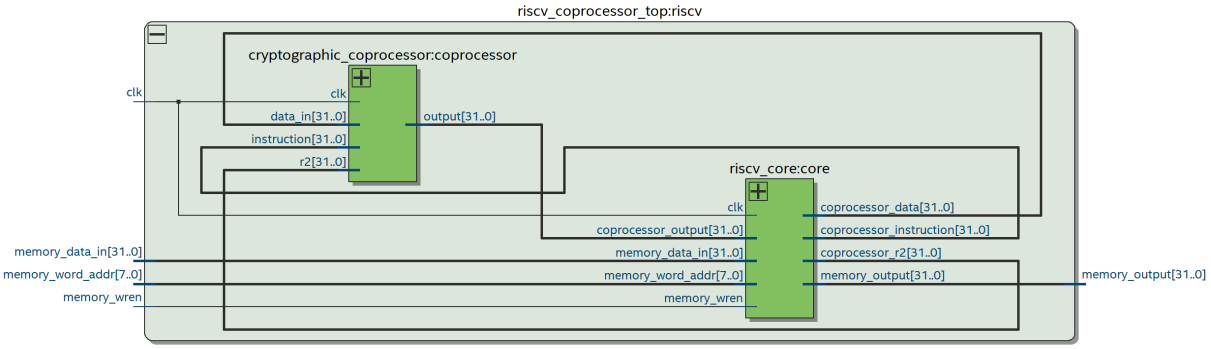

Figura 3.3: Comunicação do coprocessador criptográfico com o processador principal.

- coprocessor_output - É a saída do coprocessador, essa saída é escrita no banco de registradores de acordo com a instrução.

O coprocessador é dividido em 4 módulos principais, um para cada algoritmo de *hash* implementado, sendo eles o MD5, SHA-1, SHA-256 e SHA-512.

Os 4 módulos de *hash* funcionam de maneira similar, todos possuem um *buffer* interno que armazena um bloco a ser processado e funcionam como a máquina de estados da Figura 3.4.

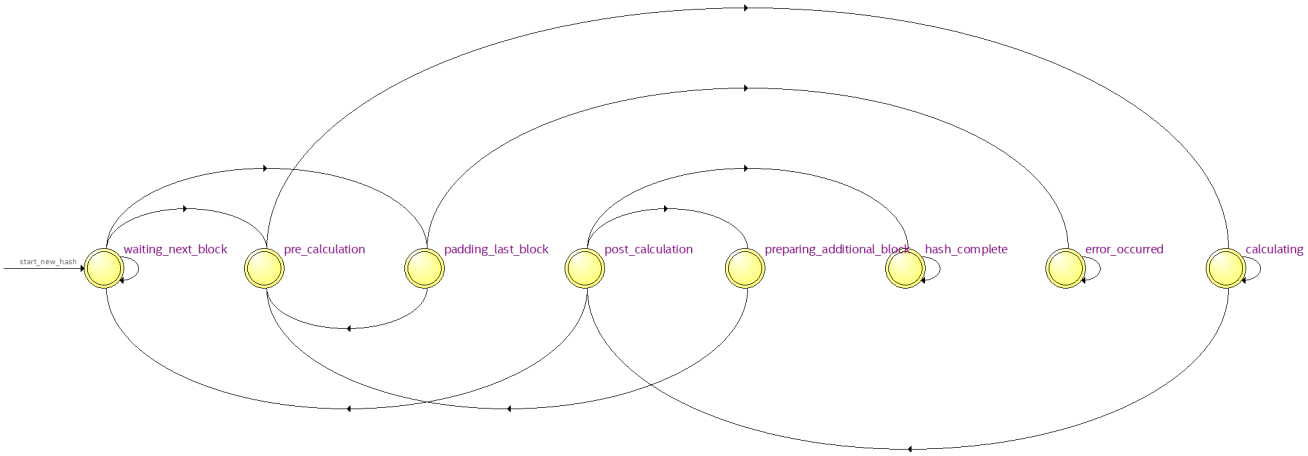

Figura 3.4: Máquina de estados do módulo de *hash* MD5.

Os estados da máquina de estados da Figura 3.4 são descritos a seguir:

waiting_next_block

Esse é o estado inicial da máquina, sempre que o módulo é resetado para calcular um novo *hash* a máquina de estados vai para esse estado. A máquina também vai para esse

Tabela 3.1: Duração do estado **calculating** de diferentes algoritmos de hash.

Algoritmo	Ciclos de clock para calcular um bloco intermediário	Ciclos de clock para calcular o último bloco da mensagem
MD5	64	64 ou 128
SHA-1	80	80 ou 160
SHA-256	64	64 ou 128
SHA-512	80	80 ou 160

estado quando algum bloco de uma mensagem já foi processado porém ainda falta outros blocos.

É nesse estado que ocorre o preenchimento do *buffer* interno do módulo com o conteúdo do próximo bloco. Após o preenchimento, podem ser dados dois comandos ao módulo, o comando para calcular o próximo bloco altera o estado para **pre_calculation** e o comando para calcular o último bloco da mensagem altera o estado para **padding_last_block**.

padding_last_block

Esse estado adiciona bits para normalizar o tamanho da mensagem de acordo com as regras do respectivo algoritmo.

Ao final desse estado, a máquina vai para o estado **pre_calculation**.

pre_calculation

Nesse estado o valor atual do *hash* é copiado para o *buffer* temporário que será utilizado e modificado no estado **calculating**.

Ao final desse estado, a máquina vai para o estado **calculating**.

calculating

É nesse estado que ocorre o processamento do próximo bloco da mensagem, durante o processamento desse estado o processador principal fica livre para executar outras instruções.

A duração desse estado varia de acordo com o algoritmo conforme mostrado na Tabela 3.1. Após esse estado a máquina muda para o estado **post_calculation**.

post_calculation

Nesse estado o *buffer* temporário do *hash* é somado ao valor anterior do *hash* e o resultado dessa soma é o novo *hash* da mensagem até o bloco atual.

A partir desse estado existem 3 próximos estado possíveis. O **hash_complete** quando o último bloco calculado é o último da mensagem e não foi necessário adicionar um bloco novo devido ao *padding*. O **preparing_additional_block** quando o último bloco calculado é o último da mensagem e é necessário adicionar um bloco extra a mensagem. E o **waiting_next_block** quando o último bloco calculado não é o último da mensagem.

preparing_additional_block

Nesse estado um novo bloco é gerado com os bits do *padding* que não couberam no bloco anterior.

Após esse estado a máquina vai para o estado **pre_calculation**.

hash_complete

Esse estado indica que o *hash* da mensagem foi concluído. A máquina fica nesse estado até ser resetada para o cálculo de um novo *hash*.

error_occurred

Esse estado indica que ocorreu um erro. A máquina fica nesse estado até ser resetada para o cálculo de um novo *hash*.

3.3 RARS

A licença *open-source* do RARS permite que qualquer pessoa baixe o programa e o modifique. As seções a seguir apresentam as modificações e funcionalidades adicionadas ao RARS para a realização desse projeto.

3.3.1 Monitor de alterações

A organização do código do RARS é dividido em 3 principais módulos (*assembler*, *tools* e *simulator*). As ferramentas (*tools*) são vários módulos independentes que não interferem diretamente na simulação, mas podem utilizar interfaces para se conectarem à simulação e serem informados de certos eventos, como acesso à memória ou aos registradores.

Para utilizar o RARS como modelo de referência nos testes que serão abordados no Capítulo 4, foi desenvolvida uma ferramenta que observa eventos de escritas nos registradores e na memória e quando um evento desse tipo ocorre, o número do registrador ou endereço de memória e o valor escrito são salvos em um arquivo.

3.3.2 Gerador de arquivo de inicialização de memória

O formato do arquivo de inicialização de memória utilizado no projeto foi o Intel HEX (.hex). A Figura 3.5 apresenta a estrutura de cada linha de um arquivo no formato Intel HEX. Os campos de cada registro são:

- *RECLEN* é o tamanho do campo *DATA*.
- *LOAD OFFSET* é o endereço do dado.
- *RECTYP* é o tipo do campo *DATA*.
- *DATA* são os dados que serão escritos na memória.
- *CHKSUM* é o complemento de dois da soma dos bits nos campos anteriores, ele é utilizado para detectar erros de transmissão.

RECORD MARK ':'	RECLEN	LOAD OFFSET	RECTYP '00'	DATA	CHKSUM
1-byte	1-byte	2-bytes	1-byte	n-bytes	1-byte

Figura 3.5: Representação de dados no formato Intel HEX (Fonte: [22]).

O problema do arquivo HEX gerado pela versão original do RARS é que o campo *LOAD OFFSET* são bytes de deslocamento do início da memória, porém o Quartus II considera que esse campo são words de deslocamento e no caso do RISC-V RV32I implementado cada word possui 32 bits ou 4 bytes.

Para resolver o problema de inicialização de memória na FPGA, o gerador de arquivos HEX do RARS foi modificado para gerar *LOAD OFFSET* como sendo deslocamento de words de 32 bits e o cálculo do *CHKSUM* também deve de ser alterado para gerar o valor correto do novo endereçamento.

3.3.3 Instruções do coprocessador

Conforme pode ser observado na Figura 3.6, o RARS também foi modificado para reconhecer e montar corretamente as instruções de hash criptográfico adicionadas nesse projeto. Para mais informações sobre as instruções adicionadas consulte o Apêndice A.

3.3.4 Contador de ciclos

Para realizar as comparações de desempenho do coprocessador implementado com o código do algoritmo de *hash* compilado, foi desenvolvida uma ferramenta que estima o

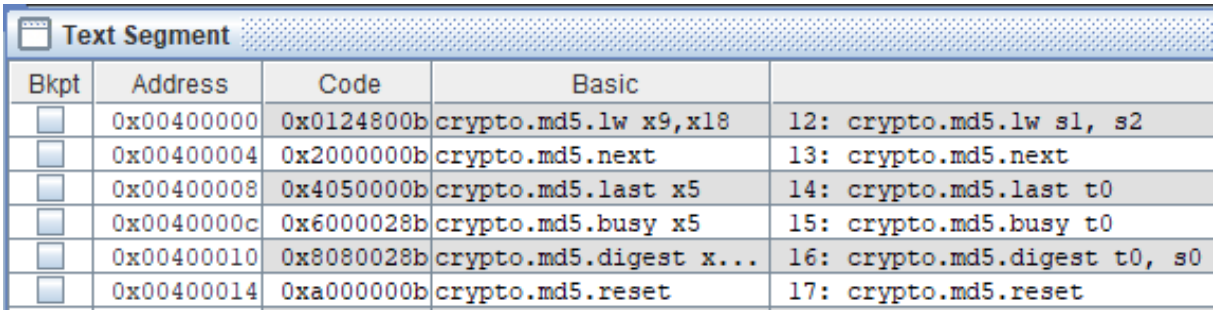

Figura 3.6: Instruções do coprocessador montadas no RARS.

número de ciclos que seriam necessários para executar o programa em uma arquitetura *pipeline*.

Essa ferramenta considera uma arquitetura *pipeline* de 5 estágios similar à descrita no livro *Computer Organization and Design RISC-V Edition: The Hardware Software Interface* [13], sem adiantamento de dados e atraso de 1 ciclo de *clock* sempre que ocorre um salto. Além disso também é adicionada uma latência de 4 ciclos de *clock* para a primeira instrução terminar de ser executada.

Capítulo 4

Verificação

Estatisticamente cerca de 70% do trabalho em um projeto de desenvolvimento de *hardware* é dedicado a verificação [23].

O processo de verificação segue o fluxo inverso à implementação. A implementação de um projeto de *hardware* normalmente consiste em partir de uma especificação bem definida e gerar módulos que idealmente cumprem as especificações. O processo de verificação parte dos módulos desenvolvidos e verifica se eles realmente cumprem as especificações [23].

O método mais utilizado para verificação de projetos implementados em VHDL é a utilização de bancadas de testes (*test benches*), porém esse método de testes normalmente requer muito trabalho manual do projetista e consequentemente consome muito tempo, além de ser muito propenso a erros.

Para simplificar a adição de novos testes e automatizar sua execução, foi utilizado *framework* VUnit. Os testes foram separados em testes unitários (*unit tests*) que testam a funcionalidade de um módulo isolado e testes de integração (*integration tests*) que testam a funcionalidade de vários módulos em conjunto.

Os testes também foram separados em testes das funcionalidades do processador RISC-V e testes do coprocessador criptográfico.

4.1 Testes unitários do processador principal

A Figura 4.1 mostra o resultado dos testes unitários do processador principal.

Foram desenvolvidos testes apenas para dois módulos do processador principal, um dos testes verifica o funcionamento do decodificador de imediatos para várias entradas diferentes e o outro teste verifica a leitura e escrita no banco de registradores.

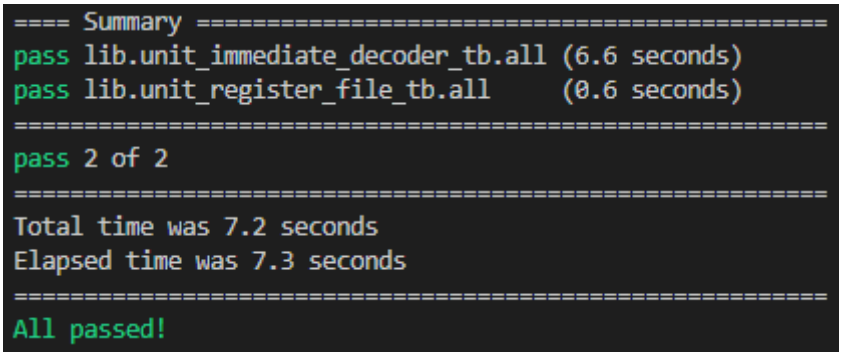

Figura 4.1: Resultado dos testes unitários do processador principal.

4.2 Testes unitários do coprocessador criptográfico

A Figura 4.2 mostra o resultado dos testes unitários do coprocessador.

Foram desenvolvidos 7 testes unitários para cada módulo de *hash* do coprocessador, totalizando 28 testes.

Esses testes verificam cada passo do algoritmo de *hash* e caso ocorra algum erro ele indica exatamente em qual passo o erro ocorreu, ajudando muito na correção de possíveis erros.

Para gerar os valores de referência do *hash* após cada passo foram utilizados os códigos em C disponibilizados nas RFCs de definição dos algoritmos [15, 18, 19], com modificações para escreverem em um arquivo o valor de cada palavra do *hash* após cada passo do algoritmo.

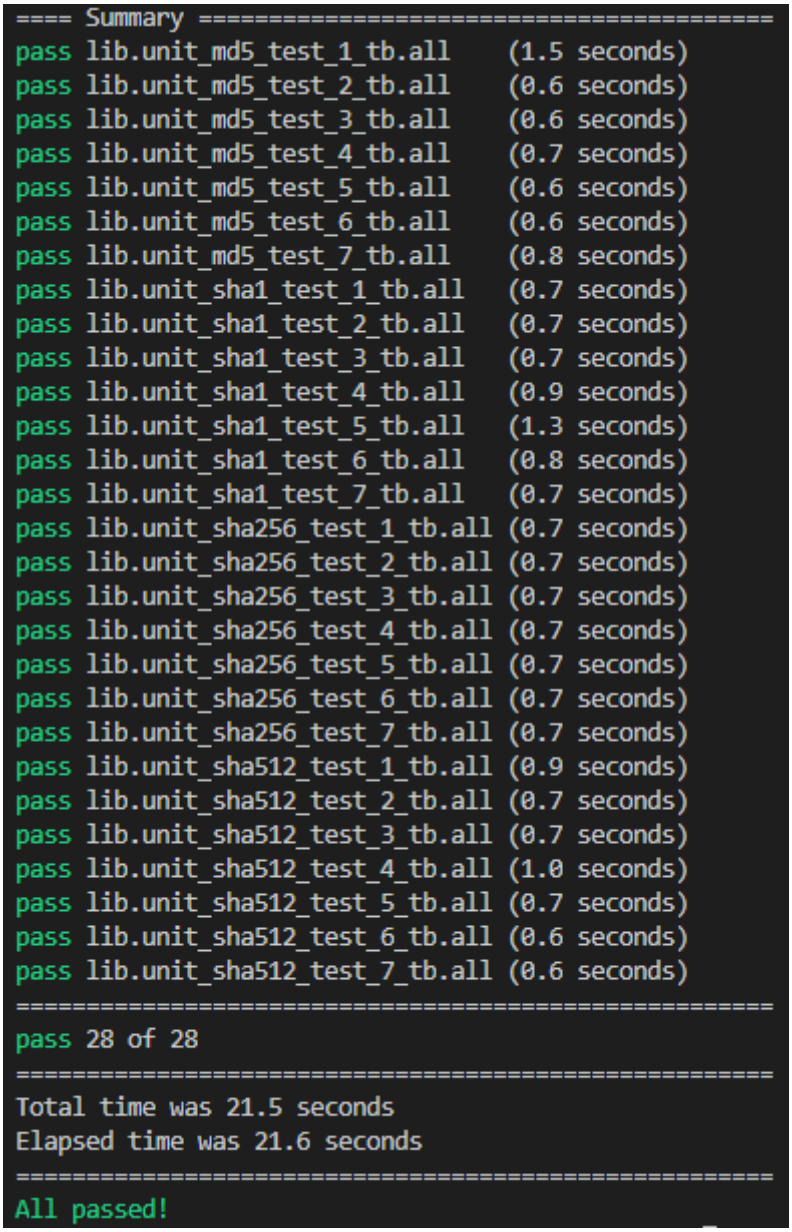

Figura 4.2: Resultado dos testes unitários do coprocessador.

4.3 Testes de integração do processador principal

A Figura 4.3 mostra o resultado dos testes de integração do processador principal.

Foram desenvolvidos 6 testes de integração para o processador principal, cada um deles testa certos tipos de instruções ou executam algoritmos conhecidos como a busca binária em um vetor ou o cálculo dos 20 primeiros números da sequencia de *fibonacci*.

A Figura 4.4 mostra o fluxo dos testes de integração do processador principal. Cada teste é baseado em um programa *assembly* qualquer, esse programa é executado uma vez no RARS, com a ferramenta Monitor de Alterações (Seção 3.3.1), e com isso é ge-

```
==== Summary ============================================
pass lib.integration_tb.simple_add    (1.7 seconds)
pass lib.integration_tb.test_1        (4.8 seconds)
pass lib.integration_tb.fibonacci     (4.2 seconds)
pass lib.integration_tb.binary_search (4.8 seconds)
pass lib.integration_tb.branches      (4.7 seconds)
pass lib.integration_tb.loop          (0.8 seconds)
=========================================================
pass 6 of 6
=========================================================
Total time was 20.9 seconds
Elapsed time was 20.9 seconds
=========================================================
All passed!
```

Figura 4.3: Resultado dos testes de integração do processador principal.

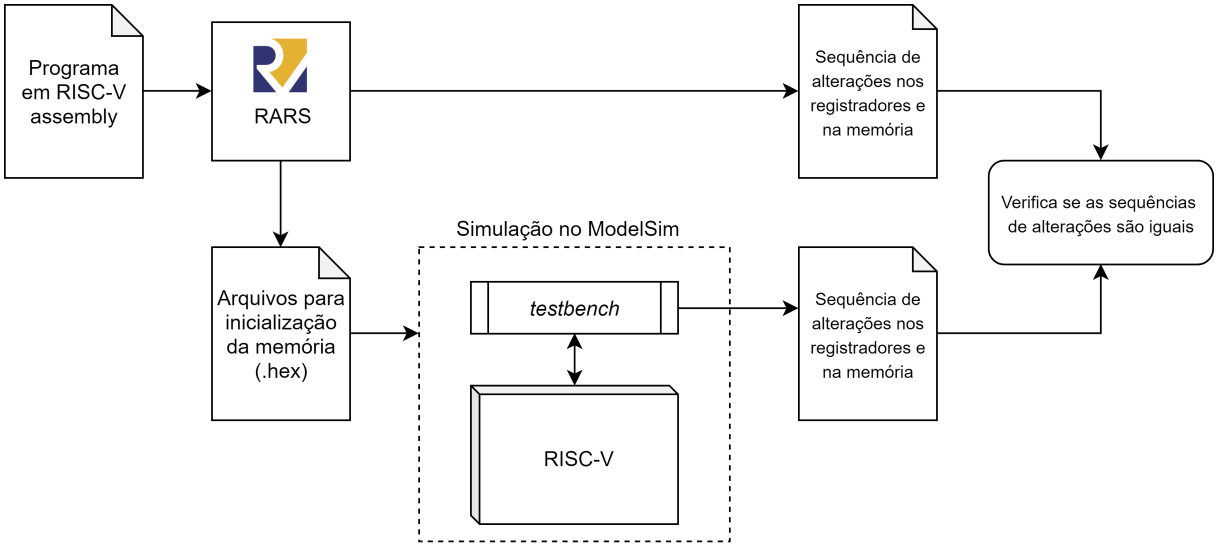

Figura 4.4: Fluxo dos testes de integração do processador principal.

rada a sequência de alterações nos registradores e na memória que será considerada a sequência correta para o programa. Ainda utilizando o RARS, são gerados os arquivos de inicialização da memória de dados e de instruções para o programa.

Com esses arquivos gerados, então o teste é adicionado no *script* do VUnit, que ao ser executado irá rodar a simulação do RISC-V com o programa carregado na memória no ModelSim. Durante a simulação o *testebench* observa qualquer alteração no banco de registradores e na memória de dados e escreve em um arquivo informações de todas as alterações que ocorreram.

Após a simulação, o VUnit chama um *script* Python que confere se o arquivo gerado na simulação é idêntico ao arquivo gerado pelo RARS e informa o resultado ao VUnit.

A grande vantagem desse tipo de teste é que a verificação roda de maneira automatizada, sem a necessidade de alguém ficar conferindo.

Além disso, esses testes foram desenvolvidos de modo que todos os testes utilizam o mesmo *testbench*, que recebe como parâmetros genéricos todas as informações necessárias para executar um teste específico. Com isso fica simples adicionar novos testes, sendo necessário apenas executá-los uma vez no RARS e informar no *script* do VUnit o nome do novo teste, o restante é feito de maneira automatizada.

4.4 Testes de integração do coprocessador criptográfico

A Figura 4.5 mostra o resultado dos testes de integração do coprocessador.

```
==== Summary ==========================================================================
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_0  (2.2 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_1  (1.2 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_2  (0.9 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_3  (1.4 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_4  (1.2 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_5  (1.6 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_6  (2.1 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_7  (2.8 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_8  (4.7 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_9  (4.6 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_10 (5.3 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_11 (5.8 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_12 (6.3 seconds)
pass lib.coprocessor_integration_tb.cryptographic_coprocessor_test_13 (6.3 seconds)
==========================================================================================
pass 14 of 14
==========================================================================================
Total time was 46.3 seconds
Elapsed time was 46.4 seconds
==========================================================================================
All passed!
```

Figura 4.5: Resultado dos testes de integração do coprocessador.

Todos os testes do coprocessador utilizam o mesmo *testbench* e o mesmo programa *assembly*. A Figura 4.6 mostra uma visão geral do fluxo de dados nesses testes.

Os testes são definidos como objetos no *script* Python do VUnit, cada um contendo uma mensagem e os valores dos *hashes* MD5, SHA-1, SHA-256 e SHA-512. Sempre que os testes são executados o *script* Python gera um arquivo no diretório da simulação com os valores no objetos em um formato que será lido pelo *testbench*.

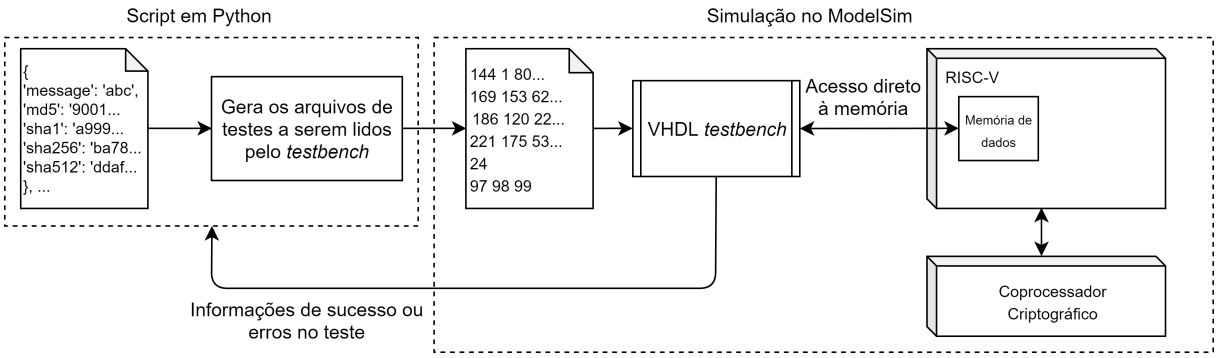

Figura 4.6: Fluxo dos testes de integração do coprocessador.

No início da simulação tanto o *testbench* e o processador RISC-V com coprocessador criptográfico são inicializados e começam a executar. Como o *testbench* tem de copiar toda a mensagem do arquivo para a memória de dados para que o cálculo do *hash* possa iniciar, a primeira parte do código *assembly* utilizado nesse teste é um *loop* que lê continuamente o endereço zero da memória de dados e só sai do *loop* quando o valor nesse endereço muda para 1.

Assim o *testbench* escreve a mensagem e o tamanho da mensagem em endereços pré determinados da memória de dados e então escreve 1 no endereço zero, nesse momento o programa em *assembly* começa a calcular os *hashes* com o auxilio do coprocessador e o *testbench* entra um *loop* esperando que o valor do endereço zero da memória mude para o valor 3.

Quando o programa *assembly* termina de calcular os *hashes*, ele escreve os resultados em endereços pré definidos, escreve o valor 3 no endereço zero e entra em um *loop* infinito que não faz mais nada. Quando o *testbench* identifica o valor 3 no endereço zero, ele verifica se os resultados escritos na memória são idênticos aos resultados esperados no arquivo gerado pelo *script* Python e informa o VUnit o resultado do teste.

Capítulo 5

Resultados

5.1 Desempenho

Para avaliar o desempenho do coprocessador desenvolvido, foi comparado o número de ciclos de *clock* necessários para calcular o *hash* de mensagens de diferentes tamanhos.

Para calcular o número de ciclos necessários pelo algoritmo original, foi utilizado o código em C disponibilizados nas RFCs de definição dos algoritmos de *hash* [15, 18, 19]. A compilação foi feita com o gcc *cross compiler* para RV32I disponível no RISC-V GNU Toolchain (Seção 2.4.4).

Para executar o código *assembly* compilado foi utilizado o simulador RARS, porém o código compilado não é diretamente suportado pelo RARS e por isso foi necessário realizar algumas alterações no código, como por exemplo, substituir as instruções que utilizam a notação *%lo(symbol)* e *%hi(symbol)* que não são suportadas pelo RARS.

Como o RARS não possui nenhuma ferramenta para estimar o número de ciclos necessários para executar um programa em uma arquitetura *pipeline*, nesse projeto foi desenvolvida uma ferramenta para esse propósito, descrita em mais detalhes na Seção 3.3.4. Essa ferramenta foi utilizada para contar os ciclos necessários para executar o código compilado dos algoritmos com um mensagem de 24 bits e os resultados estão na Tabela 5.1.

Tabela 5.1: Números de ciclos de *clocks* necessários para calcular o *hash* de uma mensagem de 24 bits com o código compilado com o gcc e simulado no RARS.

	Ciclos de clock
MD5	11978
SHA-1	21411
SHA-256	28797
SHA-512	186213

Tabela 5.2: Números de ciclos de *clocks* necessários para calcular o *hash* de uma mensagem de 24 bits no coprocessador.

	Ciclos ocupando o processador principal	Ciclos em espera	Total de ciclos
MD5	69	71	140
SHA-1	76	86	162
SHA-256	97	71	168
SHA-512	150	89	239

Para medir o número de ciclos de *clock* necessários para calcular o *hash* utilizando o coprocessador desenvolvido nesse projeto, foi considerado o número de ciclos necessários para carregar a mensagem da memória, copiar para o coprocessador e escrever o resultado final na memória. Os resultados desse teste estão na Tabela 5.2.

Como pode ser observado na Tabela 5.2, uma parte considerável do cálculo é realizado nos módulos do coprocessador e durante esse tempo o processador principal pode ficar em um *loop* esperando o resultado ficar pronto ou executar outras instruções enquanto o bloco da mensagem está sendo processado pelo módulo do coprocessador.

A Figura 5.1 apresenta uma comparação do número de ciclos necessários para calcular o SHA-1 de mensagens de diferentes tamanhos, nela podemos observar que o coprocessador apresenta um ganho enorme em relação ao código compilado e a diferença entre os dois aumenta muito conforme o tamanho da mensagem aumenta.

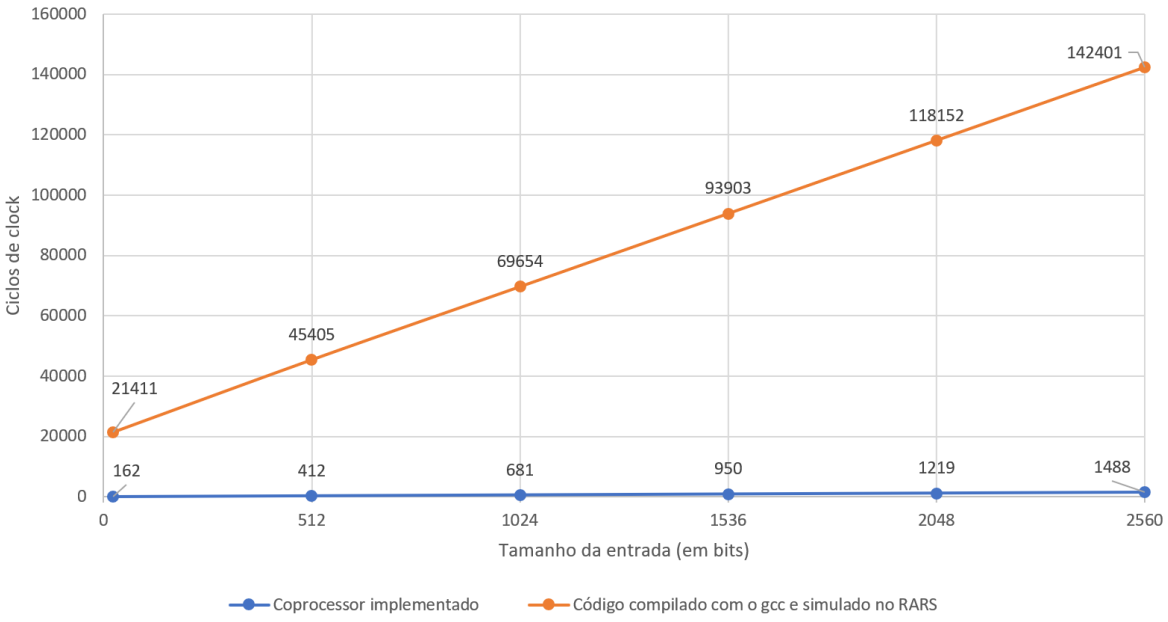

Figura 5.1: Comparação do número de ciclos necessários para calcular o SHA-1 com o coprocessador e com o código compilado.

Tabela 5.3: Utilização de recursos da FPGA DE-115.

	Apenas o processador principal	Processador principal e coprocessador
Número de elementos lógicos	3469	48244
Número de registradores	1345	14174
Número de bits de memória	16384	16384

5.2 Utilização de recursos e temporização

Utilizando o síntese do Quartus II para a FPGA DE-115, podemos observar os custos do coprocessador.

Na Tabela 5.3 podemos observar a utilização de recursos da implementação apenas do processador principal e do processador principal com o coprocessador. O número de elementos lógicos teve um aumento de quase 14 vezes e o número de registradores teve um aumento de aproximadamente 10.5 vezes ao adicionar o coprocessador de *hash* criptográfico. O número de bits de memória se manteve o mesmo.

Utilizando o Timing Analyser disponível no Quartus II, foi possível avaliar os impactos do coprocessador na temporização.

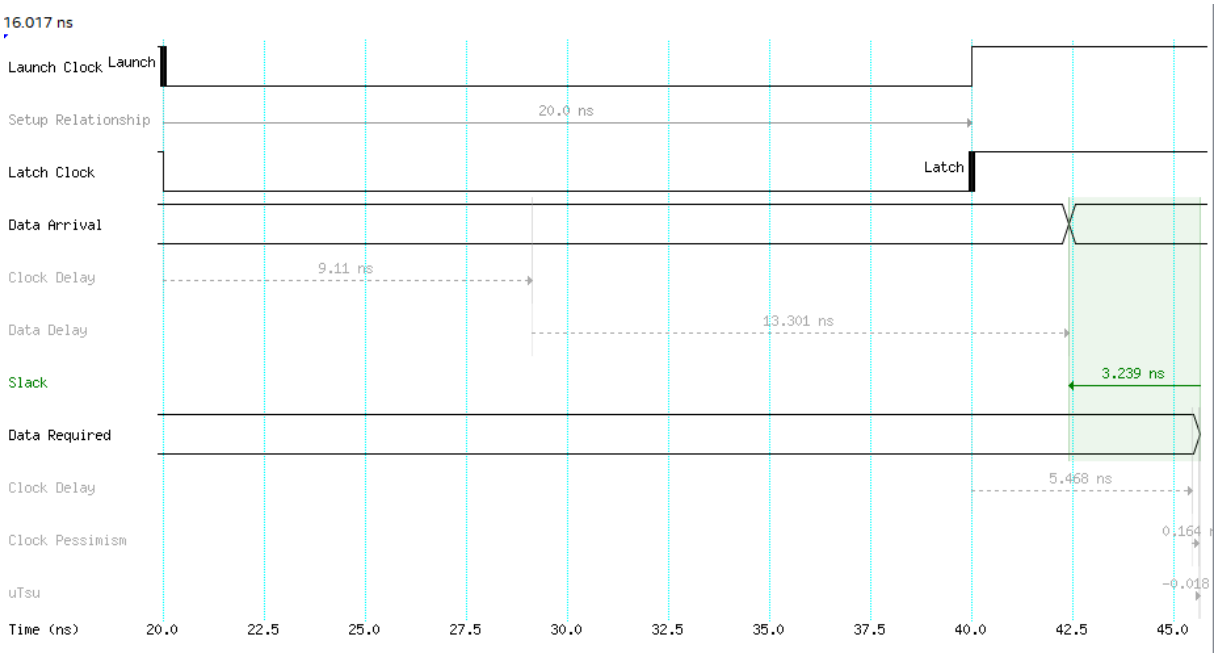

Figura 5.2: *Setup slack* do processador com o coprocessador e um clock de 25MHz.

Tabela 5.4: Comparação dos *slacks* do pior caso reportado pelo Timing Analyser, com um *clock* de 25 MHz.

	Apenas o processador principal	Processador principal e coprocessador
Setup slack	3.778 ns	3.239 ns
Hold slack	0.146 ns	0.149 ns
Recovery slack	N/A	28.125 ns
Removal slack	N/A	2.891 ns
Minimum pulse width	19.613 ns	19.593

Tabela 5.5: Comparação da frequência máxima do *clock*.

	Apenas o processador principal	Processador principal e coprocessador
Frequência máxima do clock	30.86 MHz	29.83 MHz

A Tabela 5.5 mostra as frequências máximas do *clock* reportadas pelo Timing Analyser para apenas o processador principal e para o processador com o coprocessador, nela podemos observar que a adição do coprocessador teve pouco impacto na frequência máxima do *clock*.

Como a *FPGA* DE-115 fornece apenas um *clock* de 50 MHz, foi utilizado um divisor de frequência para gerar um *clock* de 25 MHz que cumpre, com certa folga, os requisitos de temporização.

O *slack* é o tempo de folga entre o momento máximo em que um dado pode ser alterado antes ou após de ser utilizado e o momento em que o dado é alterado na implementação com um *clock* especificado. Como pode ser visto na Figura 5.2, o *slack* é o tempo de 3.239 nanosegundos em que o dado chega antes do necessário.

Um *slack* positivo significa que a implementação cumpre os requisitos de temporização e um *slack* negativo significa que existem problemas de temporização e ao sintetizar o projeto em um circuito real, provavelmente ocorrerá erros no funcionamento. O *slack* depende diretamente do período do *clock* e conforme o período do *clock* aumenta, os *slacks* diminuem.

A Tabela 5.4 mostra um resumo dos *slacks* do pior caso reportado pelo Timing Analyser.

5.3 Síntese na FPGA

Para demostrar o funcionamento da implementação desenvolvida nesse projeto, o código foi sintetizado uma FPGA e utilizado para calcular vários *hashes* da *string* "RISC-V". Os resultados desses *hashes* são mostrados nas Figuras 5.3 a 5.6.

Figura 5.3: Resultado do MD5 de *"RISC-V"* na FPGA.

Figura 5.4: Resultado do SHA-1 de *"RISC-V"* na FPGA.

Figura 5.5: Resultado do SHA-256 de *"RISC-V"* na FPGA.

Figura 5.6: Resultado do SHA-512 de *"RISC-V"* na FPGA.

Capítulo 6

Conclusão

Este trabalho apresentou uma implementação funcional de um processador com o conjunto de instruções RV32I e um coprocessador dedicado para *hash* criptográfico.

Os módulos de *hash* desenvolvidos apresentaram um ganho de performance significativo comparado com o código compilado dos algoritmos e mesmo com o custo extra de *hardware* do coprocessador, em certas aplicações que realizam muitos cálculos de *hash*, como por exemplo, na validação de transações de criptomoedas, o custo extra do *hardware* se justificaria pelo ganho da desempenho.

Durante o desenvolvimento desse trabalho também foram implementadas várias ferramentas que podem ser utilizadas em outros projetos, como por exemplo, o Monitor de Alterações do RARS e os *testbenches* genéricos desenvolvidos podem ser utilizados na validação de outras implementações do RISC-V.

Além disso, este trabalho pode ser utilizado como base para o desenvolvimento de vários trabalhos futuros como:

- Adicionar extensões opcionais do RISC-V ao processador principal.
- Adicionar suporte a palavras de 64 e 128 bits.
- Adicionar novas funcionalidades e instruções ao coprocessador.
- Realizar possíveis otimizações nos módulos de *hash* com o objetivo de reduzir o custo do *hardware* do coprocessador.
- Implementar os módulos de *hash* do coprocessador com uma arquitetura *pipeline*, o que poderia proporcionar um ganho de aproximadamente 7 vezes no tempo de cálculo dos *hashes* [24], porém essa arquitetura é extremamente complexa e provavelmente aumentaria consideravelmente o custo do *hardware*.

Referências

[1] *Open source hardware association*. `https://www.oshwa.org`, acesso em 2019-07-03. 1, 2

[2] *Open source initiative*. `https://opensource.org`, acesso em 2019-07-03. 1

[3] Gagan Gupta, Tony Nowatzki, Vinay Gangadhar e Karthikeyan Sankaralingam: *Open-source hardware: Opportunities and challenges*. Relatório Técnico, Microsoft Research and University of Wisconsin - Madison. 1

[4] Asanović, Krste e David A. Patterson: *Instruction sets should be free: The case for risc-v*. Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report(UCB/EECS-2014-146), aug 2014. 2

[5] *Risc-v foundation*. `https://riscv.org`, acesso em 2019-07-03. 2

[6] Waterman, Andrew Shell: *Design of the risc-v instruction set architecture*. Relatório Técnico, 2016. 2

[7] Patterson, David e Andrew Waterman: *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, primeira edição, nov 2017. 2

[8] Waterman, Andrew, Krste Asanovic e SiFive Inc.: *The risc-v instruction set manual*. Volume I: User-Level ISA(Document Version 2.2), may 2017. 2, 4, 5, 6, 46

[9] Hofemeier, Gael e Robert Chesebrough: *Introduction to intel® aes-ni and intel® secure key instructions*. Intel, jul 2012. 2

[10] Gulley, Sean, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford e Gil Wolrich: *Intel® sha extensions: New instructions supporting the secure hash algorithm on intel® architecture processors*. Intel® White Papers, jul 2013. 2

[11] *Arm® cortex®-a75 core cryptographic extension technical reference manual*. `http://infocenter.arm.com/help/topic/com.arm.doc.100458_0200_00_en/index.html`, acesso em 2019-07-03. 2

[12] Pala, Davide: *Design and programming of a coprocessor for a risc-v architecture*. Tese de Mestrado, Politecnico Di Torino - Collegio di Ingegneria Informatica, del Cinema e Meccatronica, 2016-2017. 5

[13] Patterson, David A. e John L. Hennessy: *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edição, 2017, ISBN 0128122757, 9780128122754. 6, 7, 8, 30

[14] Alfred J. Menezes, Paul C. van Oorschot e Scott A. Vanstone: *Handbook of Applied Cryptography*. CRC Press, dec 1996. 9

[15] Rivest, Ronald L.: *The md5 message-digest algorithm*. Rfc 1321, April 1992. `http://www.rfc-editor.org/rfc/rfc1321.txt`, acesso em 2019-07-03. 9, 10, 12, 32, 37

[16] Turner, S. e L. Chen: *Updated security considerations for the md5 message-digest and the hmac-md5 algorithms*. Rfc 6151, March 2011. `http://www.rfc-editor.org/rfc/rfc6151.txt`, acesso em 2019-07-03. 9

[17] Aumasson, Jean Philippe: *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, nov 2017. 9

[18] Eastlake, D. e P. Jones: *Us secure hash algorithm 1 (sha1)*. Rfc 3174, September 2001. `http://www.rfc-editor.org/rfc/rfc3174.txt`, acesso em 2019-07-03. 10, 12, 32, 37

[19] Eastlake, D. e T. Hansen: *Us secure hash algorithms (sha and hmac-sha)*. Rfc 4634, July 2006. `http://www.rfc-editor.org/rfc/rfc4634.txt`, acesso em 2019-07-03. 10, 15, 17, 18, 20, 32, 37

[20] Landers, Benjamin: *Rars – risc-v assembler and runtime simulator*. `https://github.com/TheThirdOne/rars`, acesso em 2019-07-03. 21

[21] Vollmar, Kenneth e Pete Sanderson: *Mars: An education-oriented mips assembly language simulator*. SIGCSE '06 Proceedings of the 37th SIGCSE technical symposium on Computer science education, páginas 239–243, mar 2006. 21

[22] *Hexadecimal object file format specification*. Relatório Técnico Revision A, Intel Corporation, jan 1988. 29

[23] Lam, William K.: *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, volume 1 edition. Prentice Hall, 2005. 31

[24] Jae-woon Kim, Hu-ung Lee e Youjip Won: *Design for high throughput sha-1 hash function on fpga*. Em *2012 Fourth International Conference on Ubiquitous and Future Networks (ICUFN)*, páginas 403–404, July 2012. 43

Apêndice A

Xcrypto1p0

Seguindo o padrão de nomenclatura de extensões do RISC-V definido no manual do conjunto de instruções [8], o nome da extensão desenvolvida nesse projeto é Xcrypto1p0, onde o X no nome da extensão indica que ela não é uma extensão padrão do RISC-V, *crypto* é o nome da extensão e 1p0 indica que a versão da extensão é a 1.0.

O código *assembly* com instruções dessa extensão pode ser montado utilizando a versão modificada do RARS disponível em `www.github.com/gustavohfc/rars`.

A Tabela A.1 apresenta o formato das instruções, a seção A.1 explica o uso de cada uma das instruções e a seção A.2 mostra um código exemplo de como utilizar as instruções dessa extensão.

Tabela A.1: Conjunto de instruções da extensão Xcrypto1p0.

funct3 [31:29]	[28:25]	rs2 [24:20]	rs1 [19:15]	[14]	funct2 [13:12]	rd [11:7]	opcode [6:0]	
000	-	rs2	rs1	-	00	-	0001011	crypto.md5.lw
001	-	-	-	-	00	-	0001011	crypto.md5.next
010	-	rs2	-	-	00	-	0001011	crypto.md5.last
011	-	-	-	-	00	rd	0001011	crypto.md5.busy
100	-	rs2	-	-	00	rd	0001011	crypto.md5.digest
101	-	-	-	-	00	-	0001011	crypto.md5.reset
000	-	rs2	rs1	-	01	-	0001011	crypto.sha1.lw
001	-	-	-	-	01	-	0001011	crypto.sha1.next
010	-	rs2	-	-	01	-	0001011	crypto.sha1.last
011	-	-	-	-	01	rd	0001011	crypto.sha1.busy
100	-	rs2	-	-	01	rd	0001011	crypto.sha1.digest
101	-	-	-	-	01	-	0001011	crypto.sha1.reset
000	-	rs2	rs1	-	10	-	0001011	crypto.sha256.lw
001	-	-	-	-	10	-	0001011	crypto.sha256.next
010	-	rs2	-	-	10	-	0001011	crypto.sha256.last
011	-	-	-	-	10	rd	0001011	crypto.sha256.busy
100	-	rs2	-	-	10	rd	0001011	crypto.sha256.digest
101	-	-	-	-	10	-	0001011	crypto.sha256.reset
000	-	rs2	rs1	-	11	-	0001011	crypto.sha512.lw
001	-	-	-	-	11	-	0001011	crypto.sha512.next
010	-	rs2	-	-	11	-	0001011	crypto.sha512.last
011	-	-	-	-	11	rd	0001011	crypto.sha512.busy
100	-	rs2	-	-	11	rd	0001011	crypto.sha512.digest
101	-	-	-	-	11	-	0001011	crypto.sha512.reset

A.1 Manual das instruções

crypto.md5.busy rd

Escreve o valor 1 no registrador **rd** se o módulo MD5 do coprocessador estiver ocupado e 0, caso contrário.

---

crypto.md5.digest r2, rd

O valor no registrador **r2** especifica uma palavra do *message digest* do MD5 que será escrita no registrador **rd**, onde a palavra mais significativa é a 0 e a menos significativa é a 3.

$0 \leq registradores[r2] \leq 3$

---

crypto.md5.last r2

Inicia o cálculo do hash do último bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo MD5 utilizando a instrução crypto.md5.lw.

O valor no registrador **r2** especifica o tamanho do último bloco em bits.

$0 \leq registradores[r2] \leq 512$

---

crypto.md5.lw r1, r2

Carrega uma palavra da memória e escreve no buffer interno do módulo MD5. O valor no registrador **r1** especifica o endereço de memória que deverá ser lido e o valor no registrador **r2** especifica a posição no buffer que a palavra será escrita, onde a palavra mais significativa é a 0 e a menos significativa é a 15.

$0 \leq registradores[r2] \leq 15$

---

crypto.md5.next

Inicia o cálculo do hash do próximo bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo MD5 utilizando a instrução crypto.md5.lw.

---

crypto.md5.reset

Reinicia todas as variáveis de estado internas do módulo MD5 para iniciar o cálculo do hash de uma nova mensagem.

crypto.sha1.busy rd

Escreve o valor 1 no registrador **rd** se o módulo SHA-1 do coprocessador estiver ocupado e 0, caso contrário.

---

crypto.sha1.digest r2, rd

O valor no registrador **r2** especifica uma palavra do *message digest* do SHA-1 que será escrita no registrador **rd**, onde a palavra mais significativa é a 0 e a menos significativa é a 4.

$0 \leq registradores[r2] \leq 4$

---

crypto.sha1.last r2

Inicia o cálculo do hash do último bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo MD5 utilizando a instrução crypto.sha1.lw.

O valor no registrador **r2** especifica o tamanho do último bloco em bits.

$0 \leq registradores[r2] \leq 512$

---

crypto.sha1.lw r1, r2

Carrega uma palavra da memória e escreve no buffer interno do módulo SHA-1. O valor no registrador **r1** especifica o endereço de memória que deverá ser lido e o valor no registrador **r2** especifica a posição no buffer que a palavra será escrita, onde a palavra mais significativa é a 0 e a menos significativa é a 15.

$0 \leq registradores[r2] \leq 15$

---

crypto.sha1.next

Inicia o cálculo do hash do próximo bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo SHA-1 utilizando a instrução crypto.sha1.lw.

---

crypto.sha1.reset

Reinicia todas as variáveis de estado internas do módulo SHA-1 para iniciar o cálculo do hash de uma nova mensagem.

crypto.sha256.busy rd

Escreve o valor 1 no registrador **rd** se o módulo SHA-256 do coprocessador estiver ocupado e 0, caso contrário.

---

crypto.sha256.digest r2, rd

O valor no registrador **r2** especifica uma palavra do *message digest* do SHA-256 que será escrita no registrador **rd**, onde a palavra mais significativa é a 0 e a menos significativa é a 8.

$0 \leq registradores[r2] \leq 8$

---

crypto.sha256.last r2

Inicia o cálculo do hash do último bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo SHA-256 utilizando a instrução crypto.sha256.lw.

O valor no registrador **r2** especifica o tamanho do último bloco em bits.

$0 \leq registradores[r2] \leq 512$

---

crypto.sha256.lw r1, r2

Carrega uma palavra da memória e escreve no buffer interno do módulo SHA-256. O valor no registrador **r1** especifica o endereço de memória que deverá ser lido e o valor no registrador **r2** especifica a posição no buffer que a palavra será escrita, onde a palavra mais significativa é a 0 e a menos significativa é a 15.

$0 \leq registradores[r2] \leq 15$

---

crypto.sha256.next

Inicia o cálculo do hash do próximo bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo SHA-256 utilizando a instrução crypto.sha256.lw.

---

crypto.sha256.reset

Reinicia todas as variáveis de estado internas do módulo SHA-256 para iniciar o cálculo do hash de uma nova mensagem.

crypto.sha512.busy rd

Escreve o valor 1 no registrador **rd** se o módulo SHA-512 do coprocessador estiver ocupado e 0, caso contrário.

---

crypto.sha512.digest r2, rd

O valor no registrador **r2** especifica uma palavra do *message digest* do SHA-512 que será escrita no registrador **rd**, onde a palavra mais significativa é a 0 e a menos significativa é a 15.

$0 \leq registradores[r2] \leq 15$

---

crypto.sha512.last r2

Inicia o cálculo do hash do último bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo SHA-512 utilizando a instrução crypto.sha512.lw.

O valor no registrador **r2** especifica o tamanho do último bloco em bits.

$0 \leq registradores[r2] \leq 1024$

---

crypto.sha512.lw r1, r2

Carrega uma palavra da memória e escreve no buffer interno do módulo SHA-512. O valor no registrador **r1** especifica o endereço de memória que deverá ser lido e o valor no registrador **r2** especifica a posição no buffer que a palavra será escrita, onde a palavra mais significativa é a 0 e a menos significativa é a 31.

$0 \leq registradores[r2] \leq 31$

---

crypto.sha512.next

Inicia o cálculo do hash do próximo bloco da mensagem. A mensagem deve ter sido carregada previamente no buffer interno do módulo SHA-512 utilizando a instrução crypto.sha512.lw.

---

crypto.sha512.reset

Reinicia todas as variáveis de estado internas do módulo SHA-512 para iniciar o cálculo do hash de uma nova mensagem.

A.2 Exemplo de uso

O código a seguir é um exemplo de como calcular o MD5 utilizando as instruções do coprocessador.

```
.data
        MD5_RESULT:          .space 16
        MESSAGE_LEN:          .word 48
        MESSAGE:         .byte 67 83 73 82 00 00 86 45 # RISC-V

.text
        la t0, MESSAGE_LEN

        # s0 is the remaining bits of the message
        lw s0, 0(t0)

        # s1 is the address of the next word to be loaded
        la s1, MESSAGE

        # s3 is the number of words per block (512/32)
        li s3, 16

MD5_NEXT_BLOCK:
        li s2, 0 # s2 is current number of words loaded into the coprocessor

MD5_NEXT_WORD:
        crypto.md5.lw s1, s2
        addi s0, s0, -32
        addi s1, s1, 4
        blez s0, MD5_LAST
        addi s2, s2, 1
        bne s2, s3, MD5_NEXT_WORD

        crypto.md5.next

MD5_WAIT_BLOCK_PROC:
        crypto.md5.busy t0
        bnez t0, MD5_WAIT_BLOCK_PROC
        j MD5_NEXT_BLOCK

MD5_LAST:
        # Calculate the numer of bits in the last block
        addi t0, s0, 32
        slli t1, s2, 5
        add t0, t0, t1
```

```
41
42          crypto.md5.last t0
43
44  MD5_WAIT_LAST:
45          crypto.md5.busy t0
46          bnez t0, MD5_WAIT_LAST
47
48
49      # s0 is the next word from the message digest to read
50      li s0, 0
51
52      # s1 is the number of 32 bits words in the MD5 message digest
53      li s1, 4
54
55      # s2 is the memory address to store the the next word of the message digest
56      la s2, MD5_RESULT
57
58  MD5_DIGEST_LOOP: # Read the MD5 result and save it in the memory
59          crypto.md5.digest t0, s0
60          addi s0, s0, 1
61          sw t0, 0(s2)
62          addi s2, s2, 4
63          bne s0, s1, MD5_DIGEST_LOOP
```