Universidade Federal de São Carlos – UFSCar
Centro de Ciências Exatas e de Tecnologia – CCET
Departamento de Computação – DC

Arthur Eugenio Silverio

Implementação de extensões criptográficas no processador VexRiscv

São Carlos
2025

Arthur Eugenio Silverio

Implementação de extensões criptográficas no processador VexRiscv

Trabalho de Conclusão de Curso apresentado à banca de graduação do Departamento de Computação do Centro de Ciências Exatas e de Tecnologia da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Ricardo Menotti

São Carlos
2025

, Arthur Eugenio Silverio

Implementação de extensões criptográficas no processador VexRiscv / Arthur Eugenio Silverio -- 2025.
45f.

TCC (Graduação) - Universidade Federal de São Carlos, campus São Carlos, São Carlos
Orientador (a): Ricardo Menotti
Banca Examinadora: Ricardo Menotti, Marcio Merino Fernandes, Helio Crestana Guardia
Bibliografia

1. Arquitetura de Computadores. 2. Criptografia. 3. Risc-v. I. , Arthur Eugenio Silverio. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Arildo Martins - CRB/8 7180

Esse trabalho é dedicado aos amigos que fiz no trilhar dessa caminhada, aos professores que dedicaram do seu tempo para minha formação, ao meu pai Edson, minha mãe Renata, minha irmã Lorena, e a Arielly, o meu amor.

"Não nos tornamos sábios pela lembrança de nosso passado,
mas pela responsabilidade de nosso futuro.
- George Bernard Shaw"

Resumo

Este trabalho apresenta o projeto, a implementação e a avaliação de um conjunto de instruções dedicadas à manipulação de bits pertencentes à extensão criptográfica *Zbk(b/c/x)* do padrão *RISC-V*, integradas ao processador *VexRiscv* e sintetizadas em *Field-programmable gate array* (FPGA). Para validar o funcionamento e quantificar os ganhos de desempenho, foi desenvolvido um *framework* de testes capaz de comparar sistematicamente as instruções aceleradas por *hardware* com suas versões equivalentes em *software*, medindo corretude, latência e *speedup* em diferentes vetores de entrada. Os resultados experimentais demonstram aceleração significativa nas instruções implementadas, variando de 2× em operações lógicas simples até mais de 100× em instruções de multiplicação polinomial em campos finitos. A integração das instruções ao pipeline do *VexRiscv* mostrou-se tecnicamente viável, preservando compatibilidade com o ecossistema *LiteX* e mantendo o número de ciclos por instruções em operações não relacionadas. Os resultados indicam que extensões criptográficas no padrão *RISC-V* oferecem benefícios substanciais mesmo em microarquiteturas compactas, destacando sua aplicabilidade para *Systems on a chip* (SoCs) configuráveis e energeticamente eficientes.

Palavras-chave: RISC-V; extensões criptográficas; manipulação de bits; aceleração por hardware; VexRiscv; FPGA; criptografia; arquitetura de computadores; desempenho; instruções customizadas..

Abstract

This work presents the design, implementation, and evaluation of a set of bit-manipulation instructions from the cryptographic extension *Zbk(b/c/x)* of the *RISC-V* standard, integrated into the *VexRiscv* processor and synthesized on an FPGA. To validate correctness and quantify performance improvements, a dedicated testing *framework* was developed to systematically compare the hardware-accelerated instructions with their software-based counterparts, measuring latency, functional equivalence, and overall speedup across diverse input vectors. Experimental results show substantial acceleration, ranging from 2× in simple logical operations to over 100× in polynomial multiplication instructions over finite fields. The integration into the *VexRiscv* pipeline proved technically feasible, maintaining compatibility with the *LiteX* ecosystem and preserving the number of cycles per instruction in unrelated operations. These findings indicate that cryptographic extensions in the *RISC-V* standard can deliver significant benefits even on compact microarchitectures, reinforcing their suitability for configurable and energy-efficient SoCs.

Keywords: RISC-V; cryptographic extensions; bit manipulation; hardware acceleration; VexRiscv; FPGA; cryptography; computer architecture; performance; custom instructions..

Lista de ilustrações

Lista de siglas

ASIC *Application-Specific Integrated Circuit*

CLB *Configurable Logic Block*

FPGA *Field-programmable gate array*

HDL *Hardware description Language*

ISA *Instruction set architecture*

IoT *Internet of things*

JVM *Java virtual machine*

K *Cryptography Extension Family*

LUT *Lookup Table*

MMU *Memory management unit*

NIST National Institute of Standards and Technology

PQC Criptografia Pós-Quântica

RISC *Reduced Instruction Set Computer*

RTL *Register-transfer level*

SoC *System on a chip*

VHDL *Very High-Speed Integrated Circuit Hardware Description Language*

VPN *Virtual Private Network*

Zk *Base Cryptography Extension Set*

Zbkb *Bitmanip instructions for Cryptography*

Zbkc *Carry-less multiply instructions*

Zbkx *Crossbar permutation instructions*

Zkne *AES Encryption Instructions*

Zknd *AES Decryption Instructions*

Zknh *SHA-2 Hashing Instructions*

Sumário

Capítulo 1

Introdução

Nas últimas décadas, a evolução das arquiteturas de processadores tem sido movida pela busca por maior eficiência e flexibilidade na construção de seus SoCs. Nesse sentido, a arquitetura de processadores *RISC-V* se destaca como uma *Instruction set architecture* (ISA) aberta e modular, consolidando-se como alternativa ideal para atender tanto a demandas acadêmicas quanto industriais, por ser livre de licenças e *royalties* em relação a padrões proprietários como *ARM* e *x86*. Essa característica tem sido explorada no desenvolvimento de novas soluções em *hardware* e em segurança para dispositivos com baixo consumo de energia e recursos computacionais limitados, usados em computadores de borda e *Internet of things* (IoT).

Uma das características mais notáveis do *RISC-V* é sua modularidade. Ela permite a criação de extensões específicas para diferentes domínios de aplicação, como aquelas voltadas para segurança e criptografia. As *RISC-V Cryptography Extensions* introduzem instruções dedicadas a algoritmos simétricos e assimétricos, como AES, SHA2 e SHA3, além de operações aritméticas modulares que manipulam bits em baixo nível — operações frequentemente usadas de forma repetitiva e massiva em algoritmos criptográficos. Com isso, determinadas tarefas podem ser executadas diretamente em *hardware* especializado, o que consequentemente reduz o consumo de energia e o tempo de processamento. Segundo Nguyen-Hoang et al. (2022), essa abordagem contribui para aumentar a segurança e padronizar implementações, tornando o *RISC-V* uma base sólida para sistemas IoT e embarcados que exigem proteção de dados.

A utilização do *RISC-V* em FPGAs como plataforma de experimentação é um passo essencial na validação de arquiteturas de processadores. Por serem dispositivos reconfiguráveis, as FPGAs permitem a implementação de diferentes variantes do *RISC-V* sem a necessidade de processos de fabricação específicos, reduzindo custos e tempo de desenvolvi-

mento. Essa flexibilidade torna o ambiente FPGA ideal para o estudo de microarquiteturas, otimizações de instruções e integração com periféricos customizados. Além disso, a prototipagem em FPGA possibilita a análise de desempenho e consumo de recursos de *hardware* de forma prática, sendo amplamente empregada em atividades relacionadas à arquitetura de computadores.

Entre as ferramentas modernas utilizadas para o desenvolvimento de sistemas baseados em *RISC-V*, destaca-se a linguagem *SpinalHDL* SpinalHDL (2025c), uma linguagem de descrição de *hardware* de alto nível baseada em *Scala*, uma linguagem de propósito geral que executa sobre uma *Java virtual machine* (JVM). *SpinalHDL* introduz abstrações que aumentam a legibilidade e a segurança do código, reduzindo erros comuns em linguagens tradicionais como *Verilog* e *VHDL*. Um dos projetos mais relevantes desenvolvidos com essa linguagem é o processador *VexRiscv* SpinalHDL (2025d), um núcleo *RISC-V* altamente parametrizável e otimizado para síntese em FPGA. *VexRiscv* permite ajustar características do *pipeline*, *cache*, suporte a *Memory management unit* (MMU) e extensões de instruções, oferecendo uma plataforma versátil para análise de desempenho e personalização arquitetural.

Complementando esse ecossistema, o *LiteX* (ENJOY-DIGITAL, 2025b) fornece uma estrutura modular para integração de processadores, controladores e periféricos em sistemas digitais complexos. Baseado em *Python*, *LiteX* automatiza a geração de sistemas em FPGA, gerenciando interconexões, memória e interfaces de comunicação de forma eficiente. Essa abordagem facilita a implementação de extensões personalizadas ao processador, permitindo a incorporação de novas instruções, aceleradores de *hardware* ou periféricos dedicados. A combinação entre *RISC-V*, *SpinalHDL*, *VexRiscv* e *LiteX* constitui, portanto, uma plataforma poderosa e aberta para o desenvolvimento e a experimentação de arquiteturas de processadores, contribuindo significativamente para o avanço da pesquisa e inovação em sistemas digitais reconfiguráveis.

1.1 Objetivo Geral

O objetivo primário deste trabalho é compreender o desenvolvimento de extensões do conjunto de instruções (ISA) em processadores baseados no *VexRiscv*, uma implementação altamente parametrizável capaz de executar Linux em FPGAs. Além disso, busca-se ampliar o conhecimento em arquitetura de computadores e no desenvolvimento de *frameworks* como o *LiteX*, bem como de recursos que auxiliem no desenvolvimento de extensões de processadores e periféricos que aumentem a confiabilidade e a segurança de dispositivos de borda.

1.2 Objetivos específicos

- Compreender a estrutura modular do *VexRiscv* e sua integração com o *framework LiteX*, explorando seu fluxo de desenvolvimento em FPGAs;
- Desenvolver e validar um ambiente de experimentação em FPGA para testar extensões de processador voltadas à criptografia em ambientes simulados e em hardware real, considerando também sua integração em SoCs;
- Estudar técnicas de integração de aceleradores criptográficos ao *pipeline* do processador;
- Implementar e analisar extensões de criptografia no núcleo *VexRiscv*, avaliando o impacto em desempenho, área e consumo de recursos em FPGA;
- Documentar e disponibilizar os resultados do desenvolvimento das extensões e do ambiente de teste, contribuindo para a comunidade acadêmica e de código aberto.

Capítulo 2

Fundamentação Teórica

2.1 FPGAs

Os FPGAs são dispositivos semicondutores reconfiguráveis compostos por uma matriz bidimensional de blocos lógicos interligados por uma rede de interconexão também programável. Sua construção segue uma arquitetura de blocos lógicos cercados por uma malha de roteamento reconfigurável, como descrito em Brown (1996).

Os *Configurable Logic Blocks* (CLBs) formam a unidade fundamental de computação dos FPGAs e integram principalmente as *Lookup Tables* (LUTs), elementos sequenciais capazes de implementar funções combinacionais e sequenciais de forma flexível. Além disso, possibilitam a reconfiguração completa do dispositivo após a fabricação, enquanto os multiplexadores internos são responsáveis pelo direcionamento dos sinais. A etapa de *placement*, realizada pelas ferramentas de síntese, é responsável por mapear cada operação lógica descrita em linguagens de descrição de hardware, como *Very High-Speed Integrated Circuit Hardware Description Language* (VHDL) e *Verilog*, para CLBs específicos dentro do dispositivo, enquanto o roteamento determina os caminhos físicos entre esses blocos.

Além de sua função como plataforma de implementação final, a escolha de FPGAs se justifica pela combinação de desempenho, paralelismo e capacidade de reconfiguração pós-fabricação. Enquanto as *Application-Specific Integrated Circuit* (ASICs) oferecem alta eficiência, não podem ser modificadas após a produção, e arquiteturas tradicionais como *CPUs* e *GPUs* não permitem personalização estrutural profunda. Os autores em Boutros e Betz (2021) enfatizam que essa adaptabilidade torna os FPGAs adequados para aplicações que exigem otimização específica ou prototipagem rápida. Essa característica também torna possível o desenvolvimento paralelo de *software* e *hardware*, permitindo a realização de validações funcionais, testes de arquitetura e simulações de desempenho em

estágios iniciais, reduzindo custos e acelerando o ciclo de desenvolvimento.

2.2 Linguagens de Descrição de Hardware

As *Hardware description Languages* (HDLs) são ferramentas fundamentais no desenvolvimento de circuitos digitais, pois permitem modelar o comportamento e a estrutura do *hardware* em níveis que vão desde a lógica combinacional até sistemas complexos baseados em registradores (THOMAS; MOORBY, 1991). Diferentemente das linguagens de programação tradicionais, as HDLs descrevem paralelismo, temporização e relações elétricas entre sinais, possibilitando que sintetizadores convertam essas descrições em estruturas físicas implementáveis em ASICs ou FPGAs. Entre as HDLs clássicas, destacam-se *Verilog* e VHDL, amplamente suportadas por ferramentas comerciais e de código aberto, e consideradas padrão na indústria de semicondutores.

O *Verilog* é uma das linguagens mais utilizadas em projetos digitais devido à sua sintaxe compacta, próxima de linguagens como C, e à grande compatibilidade com ferramentas de simulação e síntese. Sua evolução, o *SystemVerilog*, adiciona recursos avançados para verificação e modelagem orientada a objetos, ampliando seu uso em ambientes de teste complexos. Por outro lado, VHDL possui uma estrutura mais rígida e verbosa, o que tende a reduzir ambiguidades e favorecer projetos em setores altamente regulados, como os segmentos aeroespacial e automotivo (SHAHDAD, 1986). Ambas as linguagens seguem o paradigma descritivo tradicional, baseado em blocos concorrentes e processos sensíveis a sinais, refletindo diretamente o comportamento do *hardware*.

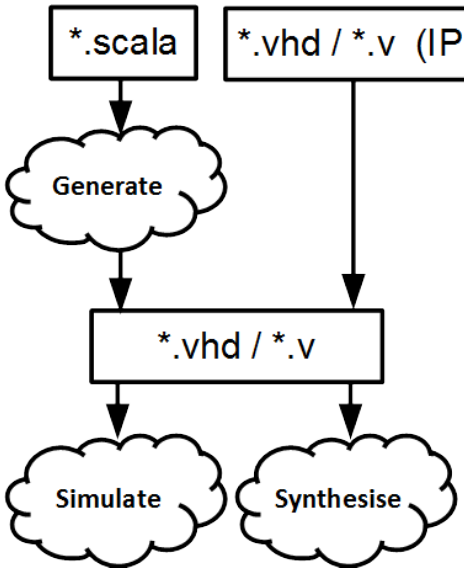
As HDLs também promovem a organização hierárquica do projeto ao permitir a construção modular, em que blocos de *hardware* podem ser definidos, instanciados e conectados com interfaces bem estruturadas. (THOMAS; MOORBY, 1991) destacam que essa abordagem modular favorece a reutilização e reduz significativamente erros estruturais durante o desenvolvimento, além de facilitar a verificação funcional e temporal por meio de simulações rigorosas.

Nos últimos anos, surgiram linguagens de descrição baseadas em linguagens de propósito geral, como *Scala*, resultando em *frameworks* modernos como *Chisel* Bachrach et al. (2012) e *SpinalHDL* SpinalHDL (2025b). Essas linguagens introduzem maior abstração, permitindo geração automática de circuitos, reutilização de componentes, parametrização extensiva e integração com ferramentas modernas de *software*. *Chisel*, desenvolvido na Universidade da Califórnia em Berkeley, permite que desenvolvedores criem circuitos como construções em *Scala*, resultando em HDL gerada automaticamente com alto nível de expressividade. *SpinalHDL* segue uma abordagem semelhante, mas com foco em produtividade e redução de *boilerplates* [1], oferecendo verificação de conexões, análise estática e geração eficiente de *Verilog* sintetizável. Essas linguagens modernas vêm ganhando es-

[1] código repetitivo e previsível

paço por facilitar o desenvolvimento de *hardware* complexo, como SoCs, aceleradores e arquiteturas *RISC-V*, ao mesmo tempo em que mantêm compatibilidade completa com fluxos tradicionais de ferramentas por meio da geração final de *Verilog* ou VHDL. A figura 1 é uma ilustração do fluxo de trabalho com HDLs usando *SpinalHDL*.

Figura 1 – Fluxo de geração de HDL usando SpinalHDL



Fonte: SpinalHDL (2025b)

2.3 Arquitetura RISC-V

A arquitetura *RISC-V* é uma ISA aberta, modular e especificada com base nos princípios de projeto *Reduced Instruction Set Computer* (RISC), conforme descrito em Waterman et al. (2025a) e Waterman et al. (2025b). Diferentemente de ISAs proprietárias, como *ARM* e *x86*, o *RISC-V* é disponibilizado sob uma licença permissiva, permitindo que qualquer pessoa projete, implemente, modifique ou comercialize processadores compatíveis sem custos de licenciamento. Essa característica torna a arquitetura especialmente atraente para pesquisa acadêmica, empresas de semicondutores, fabricantes de dispositivos embarcados e projetos de *hardware* aberto. A ISA foi projetada para ser simples, coerente e extensível, garantindo longevidade, portabilidade e eficiência tanto em implementações mínimas quanto em projetos de alto desempenho.

A arquitetura *RISC-V* é estruturada em torno de um conjunto base reduzido, ao qual extensões opcionais podem ser adicionadas conforme a necessidade. O conjunto base inteiro é definido para operações inteiras (*RV32I*, *RV64I*), enquanto extensões padronizadas fornecem capacidades adicionais, como operações atômicas (A), multiplicação e divisão (M), suporte a ponto flutuante (F e D), compactação de instruções (C) e vetorização (V). Essa modularidade permite que projetistas de processadores selecionem apenas os blocos necessários ao domínio de aplicação, resultando em implementações mais eficientes em

área, consumo e desempenho. Esse modelo também facilita o desenvolvimento de aceleradores específicos e extensões personalizadas, mantendo compatibilidade com o ecossistema *RISC-V*.

O avanço do *RISC-V* é impulsionado por uma comunidade ativa e um amplo ecossistema de ferramentas, incluindo compiladores como *GCC* e *LLVM*, simuladores, depuradores, *Linux kernels* e diversos sistemas operacionais de tempo real. No domínio de *hardware*, a arquitetura possui implementações que variam desde microcontroladores extremamente compactos até processadores de múltiplos núcleos com suporte a virtualização e vetores. Além disso, a abertura da ISA permitiu a criação de processadores altamente configuráveis, como o *VexRiscv*, *Rocket* e *NeoRV32*, muitos dos quais escritos em linguagens modernas como *Chisel* e *SpinalHDL*. A combinação de abertura, flexibilidade e suporte crescente posiciona o *RISC-V* como uma das plataformas mais promissoras para inovação em arquiteturas de *hardware*, sistemas embarcados e computação de alto desempenho.

2.4 Extensões Criptográficas do RISC-V

As extensões de criptografia do *RISC-V* foram desenvolvidas para fornecer suporte nativo a operações criptográficas de forma eficiente, segura e padronizada, reduzindo a necessidade de aceleradores externos ou rotinas otimizadas em software, possibilitando que implementações da ISA alcancem alta velocidade em algoritmos simétricos, *hashing* e aritmética de curvas elípticas. A abordagem modular segue o mesmo princípio da arquitetura *RISC-V*, permitindo que cada núcleo implemente apenas os subgrupos necessários para seu domínio de aplicação.

Algoritmos criptográficos dependem fortemente de operações repetidas e de padrão fixo, como rotações de bits, permutações, *s-boxes*, *mixes*, operações *xor*, adições modulares, multiplicações modulares, entre outras. Quando essas operações são implementadas por software, o processador precisa executar várias instruções para simular o comportamento desejado. Quando há instruções dedicadas, o *hardware* executa todo o bloco funcional de uma vez (MARSHALL; PAGE; PHAM, 2021).

Um exemplo simples desse fato pode ser visto analisando uma instrução da extensão de criptografia. Considere uma operação simples de rotação de 32 bits (*rori*), que pode ser visualizada em *C*:

```c
#define XLEN 32
__attribute__((noinline)) uint32_t emu_rori(uint32_t rs1, uint32_t imm) {
    uint32_t shamt = imm & (XLEN - 1);
    return (rs1 >> shamt) | (rs1 << (XLEN - shamt));
}
```

Ela seria compilada para o seguinte código em *assembly*, que traz várias instruções para conseguir emular o comportamento desejado:

```
lw      a5,-40(s0)        # a5 = shift
andi    a5,a5,31          # a5 = shift mod 32 (evita rotacoes invalidas)
sw      a5,-20(s0)        # salva shift_mod

lw      a5,-36(s0)        # a5 = valor
lw      a4,-20(s0)        # a4 = shift_mod
srl     a4,a5,a4          # a4 = valor >> shift_mod   (parte direita)

lw      a3,-20(s0)        # a3 = shift_mod
neg     a3,a3             # a3 = -shift_mod
andi    a3,a3,31          # a3 = (32 - shift_mod) & 31
sll     a5,a5,a3          # a5 = valor << (32 - shift_mod) (parte esquerda)

or      a5,a5,a4          # a5 = (valor >> sh) | (valor << (32 - sh))
                          #     -> rotacaoo para a direita (rotate right)
```

ou considerando algumas otimizações do compilador *-O3*:

```
andi    a1,a1,31
neg     a5,a1
sll     a5,a0,a5
srl     a0,a0,a1
or      a0,a5,a0
ret
```

A mesma operação torna-se uma única instrução em uma *CPU* que implementa as extensões de manipulação de bits:

```
    rori a0, a0, a1
```

Outro exemplo clássico é o das *s-boxes*. Com apenas uma instrução é possível executar um *MixColumns* parcial ou um *ShiftRows* parcial sem depender de acesso à memória, uso de tabelas em *RAM* ou *ROM*, cuja latência pode depender de *cache*. Tudo ocorre em uma única operação atômica implementada diretamente em *hardware* combinacional:

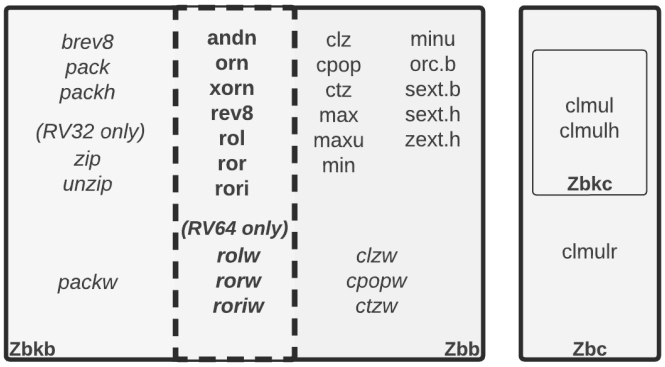
```
    aes32esmi t0, a0, a1, 0
```

A *Cryptography Extension Family* (K) é dividida em várias subextensões específicas, cada uma voltada para um conjunto distinto de operações. A subextensão *Base*

Cryptography Extension Set (Zk), núcleo da família, é subdividida em módulos como *Bitmanip instructions for Cryptography* (Zbkb) e *Carry-less multiply instructions* (Zbkc) (operações de manipulação de bits de suporte para criptografia), *Crossbar permutation instructions* (Zbkx) (operações auxiliares genéricas), *AES Encryption Instructions* (Zkne) e *AES Decryption Instructions* (Zknd) (instruções otimizadas para *AES*), e *SHA-2 Hashing Instructions* (Zknh) (instruções aceleradoras para *SHA-2*). Essas extensões introduzem instruções dedicadas para operações como *SubBytes*, *MixColumns* e expansões de chave, reduzindo significativamente a latência de implementações *AES* em software. De forma similar, as operações de *SHA-256* e *SHA-512* passam a ser executadas por instruções especializadas, que substituem blocos inteiros de software, beneficiando aplicações que dependem de *hashing* intensivo, como autenticação de *firmware*, protocolos *TLS* e assinaturas digitais.

A Figura 2 apresenta um comparativo entre as extensões de manipulação de bits e as extensões específicas para criptografia.

Figura 2 – Comparativo da extensões de manipulação de bit e criptografia



Fonte: FpRox (2023)

Há também subextensões avançadas direcionadas à criptografia de chave pública, como conjuntos específicos para aritmética modular, que servem de base para algoritmos de curvas elípticas e criptografia pós-quântica. Embora ainda estejam em desenvolvimento contínuo, essas extensões buscam padronizar operações fundamentais como multiplicação modular de grande porte e redução *Montgomery*, possibilitando implementações mais rápidas e menos propensas a falhas de segurança por *timing attacks*. A padronização dessas capacidades dentro da ISA permite que sistemas *RISC-V* atendam a requisitos modernos de segurança sem depender de *hardware* proprietário, fortalecendo o *RISC-V* como plataforma competitiva para aplicações críticas.

2.5 Core VexRiscv

VexRiscv SpinalHDL (2025d) é um núcleo de processador *RISC-V FPGA-friendly* de 32 bits altamente configurável, desenvolvido em *SpinalHDL*, e amplamente utilizado em sistemas embarcados e FPGAs devido à sua combinação de eficiência, modularidade e desempenho. Diferentemente de implementações tradicionais escritas diretamente em *Verilog* ou *VHDL*, o *VexRiscv* aproveita as capacidades de construção de *hardware* do *SpinalHDL* para gerar automaticamente o *Register-transfer level* (RTL) otimizado com grande flexibilidade estrutural. Essa abordagem possibilita que um único projeto seja instanciado em múltiplas variantes — desde microcontroladores extremamente compactos até processadores com *pipeline* profundo, caches, MMU e suporte a sistemas operacionais como *Linux* e sistemas operacionais de tempo real.

Baseado no *NaxRiscv* SpinalHDL (2025a), a principal característica do *VexRiscv* é sua arquitetura altamente parametrizável, baseada em *plugins*. Cada *plugin* adiciona uma funcionalidade específica ao núcleo, como a unidade de multiplicação e divisão, unidade de compressão de instruções (C), suporte a interrupções externas, barramentos *AXI/Avalon/Wishbone*, ou até mesmo mecanismos de depuração via *JTAG* — um padrão de indústria para teste e depuração de circuitos eletrônicos. Essa modularidade permite que o desenvolvedor escolha apenas os componentes necessários ao seu projeto, obtendo um balanço ideal entre área, consumo e desempenho. Graças a essa flexibilidade, o *VexRiscv* pode operar em configurações mínimas com poucas LUTs ou atingir frequência de *clock* elevada em variantes otimizadas para *pipelines* mais robustos.

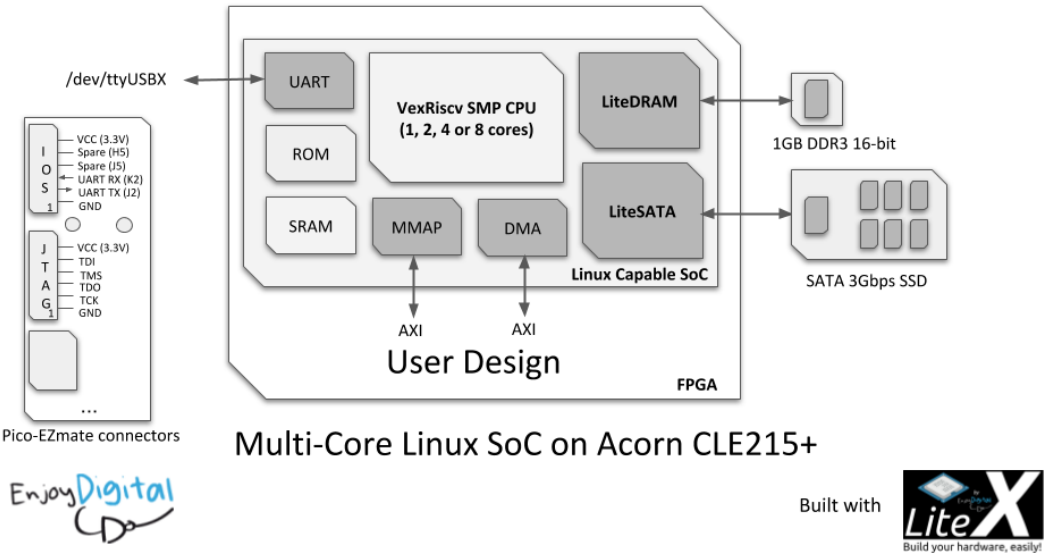
Outro destaque é o forte uso do *VexRiscv* em plataformas FPGAs e em ecossistemas de *hardware* aberto, especialmente em projetos educacionais *RISC-V*, sistemas baseados em *LiteX* e aplicações que exigem rápida prototipação. Sua eficiência é comprovada por implementações que alcançam desempenho competitivo em relação a núcleos comerciais equivalentes, mantendo compatibilidade com todo o conjunto base *RISC-V* RV32I e extensões amplamente adotadas. Além disso, a geração automática de *Verilog* sintetizável facilita sua integração com fluxos industriais e ferramentas tradicionais de *EDA*. O *VexRiscv* se consolidou como uma das implementações *RISC-V* mais versáteis e acessíveis para FPGAs, especialmente em projetos que demandam personalização profunda do *pipeline* e do conjunto funcional.

2.6 Framework LiteX

LiteX é um *framework* de *hardware* aberto voltado para a construção de SoCs modulares em FPGAs, que inclui suporte nativo a núcleos *RISC-V* como parte de sua arquitetura. Desenvolvido em *Python* sobre o ecossistema *Migen* apresentado em Kermarrec et al. (2020), *LiteX* permite gerar automaticamente sistemas complexos com barramentos,

controladores de memória, periféricos e infraestrutura de interconexão, reduzindo significativamente o esforço necessário para integrar em FPGAs um processador *RISC-V*. Dentro desse ambiente, o "LiteX RISC-V" refere-se ao conjunto de núcleos suportados — como *VexRiscv*, *PicoRV32*, *Rocket*, *NeoRV32* e outros — combinados com os subsistemas gerados pelo *framework*, formando um SoC completamente parametrizável. A figura 3 mostra um exemplo de *SoC multi-core* capaz de executar linux disponível em Enjoy-Digital (2025a).

Figura 3 – SoC Multi-core Linux Capable baseado no VexRiscv-SMP CPU, com LiteDRAM e LiteSATA



Fonte: Enjoy-Digital (2025b)

Uma característica do *LiteX* no contexto *RISC-V* é sua abordagem altamente configurável, que permite ajustar desde aspectos de microarquitetura do processador até o tipo de barramento, memória e periféricos presentes no sistema. O núcleo *RISC-V* selecionado é conectado ao sistema por meio do barramento, possibilitando adicionar módulos como controladores *UART*, *Ethernet*, *SPI*, *timers*, *GPIO* e controladores de interrupção, tudo isso sem a necessidade de escrever o RTL manualmente. Além disso, o *LiteX* oferece integração direta com módulos como *LiteDRAM*, *LiteSDCard*, *LiteEth*, *LitePCIe*, *LiteSATA* e outras implementações que fornecem recursos presentes em diversas FPGAs, permitindo que o processador execute softwares mais complexos, desde *bare-metal* até sistemas operacionais embarcados.

Outra contribuição importante do *LiteX* para implementações *RISC-V* é sua capacidade de gerar *bitstreams*, BIOS e imagens de software completas, automatizando o fluxo de desenvolvimento. O *framework* inclui ferramentas para construção da BIOS (em C), inicialização de memória, carregamento de programas via interface serial e, em configurações mais avançadas, suporte à execução de *Linux* em FPGAs com núcleos *RISC-V*

mais robustos, como variantes do *VexRiscv* com MMU. A simplicidade do fluxo, combinada à modularidade e ao suporte amplo à experimentação, faz do *LiteX* uma plataforma ideal tanto para pesquisa acadêmica quanto para prototipagem industrial, consolidando-o como uma das soluções mais flexíveis para o desenvolvimento de SoCs *RISC-V*.

2.7 Algoritmos de Criptografia

A criptografia é o campo da segurança da informação responsável por desenvolver técnicas para garantir confidencialidade, integridade e autenticidade. Seus fundamentos matemáticos são baseados em teoria dos números, álgebra abstrata, funções *hash* e estruturas probabilísticas, permitindo transformar dados legíveis em representações cifradas resistentes a ataques. De maneira geral, os sistemas criptográficos são classificados em dois grandes grupos: criptografia simétrica e criptografia assimétrica, cada um adequado a cenários e requisitos distintos de comunicação segura.

A criptografia simétrica utiliza a mesma chave para cifrar e decifrar dados. Algoritmos como *AES*, *ChaCha20* e *DES* operam sobre blocos ou fluxos de bits aplicando transformações matemáticas que dependem fortemente da expansão de chave e de operações não lineares.

A criptografia assimétrica, baseada em pares de chaves pública e privada, é construída sobre problemas matemáticos difíceis de inverter, como a fatoração de inteiros (*RSA*), o logaritmo discreto (*Diffie-Hellman*) e a aritmética de curvas elípticas (*ECDSA*, *EdDSA*). Esse modelo permite autenticação, troca segura de chaves e assinaturas digitais, integrando protocolos essenciais como *TLS*, *Virtual Private Networks* (VPNs) e sistemas de certificação.

2.8 Criptografia Pós-Quântica

A Criptografia Pós-Quântica (PQC) surgiu como resposta ao avanço previsto dos computadores quânticos, que ameaçam comprometer algoritmos assimétricos tradicionais por meio de técnicas como o algoritmo de *Shor*, capaz de resolver fatoração e logaritmos discretos de maneira exponencialmente mais rápida do que computadores clássicos. A PQC busca desenvolver algoritmos seguros mesmo na presença de adversários equipados com computadores quânticos universais, mantendo eficiência prática e compatibilidade com protocolos existentes.

Os esquemas pós-quânticos são baseados em problemas matemáticos considerados resistentes a ataques quânticos, como reticulados (*lattices*), códigos corretores de erro, funções *hash* e modelos multivariáveis não lineares. Entre os algoritmos mais promissores estão os esquemas de chave pública baseados em reticulados, como *Kyber* (criptografia/KEP) e *Dilithium* (assinaturas), ambos padronizados pelo National Institute of Stan-

dards and Technology (NIST). Esses sistemas utilizam operações em espaços vetoriais de alta dimensão, explorando problemas como *Module-LWE* e *Module-SIS*, cuja solução permanece computacionalmente inviável para computadores clássicos e quânticos conhecidos. Além disso, algoritmos como *SPHINCS+*, fundamentado exclusivamente em funções *hash*, oferecem uma alternativa extremamente robusta, embora ao custo de chaves e assinaturas maiores.

Com a padronização da PQC e seu avanço em implementações embarcadas, FPGAs, microcontroladores e sistemas *RISC-V*, a transição para algoritmos resistentes ao cenário pós-quântico tornou-se um objetivo estratégico global. Esse movimento é essencial para garantir longevidade e segurança de infraestruturas críticas, comunicação segura e proteção de dados sigilosos contra adversários que podem armazenar informações hoje e decifrá-las futuramente com capacidade quântica (*"harvest now, decrypt later"*). Assim, a criptografia pós-quântica representa um passo fundamental para a continuidade da segurança digital em médio e longo prazo.

Capítulo 3

Desenvolvimento

O presente trabalho teve como objetivo implementar e avaliar um conjunto de instruções pertencentes à extensão de criptografia do padrão *RISC-V*, integrando-as a um núcleo *RISC-V* sintetizado em FPGA e executado no ambiente *LiteX*. Para isso, foram escolhidos como plataforma de prototipação o núcleo *VexRiscv*, devido à sua elevada modularidade, e a placa *Sipeed Tang Primer 20K*, equipada com a *FPGA Gowin GW2AR-18*, que oferece recursos suficientes para exploração arquitetural. A implementação prática consistiu na modificação do pipeline do processador para incorporar instruções criptográficas dedicadas e na posterior avaliação de impacto sobre desempenho.

A extensão utilizada segue as especificações *RISC-V* destinadas a operações criptográficas, incluindo instruções para manipulação de blocos, primitivas aritméticas otimizadas, rotações, misturas e operações fundamentais para algoritmos modernos. Este trabalho concentrou-se na implementação parcial dessas instruções em hardware, priorizando aquelas relacionadas à manipulação de bits. Isso exigiu a modificação de unidades funcionais existentes do *VexRiscv* e, em alguns casos, a adição de blocos combinacionais ou sequenciais diretamente em *SpinalHDL*, garantindo compatibilidade com a arquitetura e preservação da lógica de controle do pipeline.

Após a implementação, conduziram-se *benchmarks* comparativos utilizando ferramentas integradas ao *LiteX*, bem como rotinas específicas desenvolvidas em *C* para explorar as instruções customizadas. Os experimentos foram realizados diretamente na FPGA, garantindo que os resultados refletissem atrasos reais de hardware, consumo de LUTs, *flip-flops*, blocos de memória e alterações na frequência máxima atingível após síntese e *place-and-route*. A análise final confrontou o desempenho do processador antes e depois da adição das instruções criptográficas, permitindo quantificar ganhos de performance, custos de área e eventuais limitações introduzidas no pipeline.

3.1 Execução dos Algoritmos com Aceleração Criptográfica

Durante a etapa de validação funcional, avaliou-se a viabilidade de executar algoritmos criptográficos reais utilizando bibliotecas amplamente empregadas como WolfSSL (2025) e OpenSSL (2025). O objetivo era determinar se essas bibliotecas poderiam se beneficiar das instruções aceleradas implementadas no hardware e, ao mesmo tempo, fornecer um ambiente comparável ao encontrado em sistemas embarcados reais.

A biblioteca *OpenSSL*, embora altamente madura e amplamente adotada, revelou-se inviável para a plataforma utilizada devido ao seu tamanho substancial. Mesmo versões compiladas com agressiva remoção de módulos, *link-time optimization* e configurações minimalistas ultrapassavam a capacidade de armazenamento e memória da plataforma embarcada construída sobre *LiteX*. Além disso, a arquitetura monolítica do *OpenSSL* dificulta sua adaptação para ambientes reduzidos, tornando sua integração incompatível com o fluxo experimental empregado neste trabalho.

Por outro lado, a biblioteca *WolfSSL* se mostrou mais promissora, dado seu foco explícito em ambientes embarcados e sua modularidade superior. Foi realizada uma redução agressiva da biblioteca, removendo ciphers, módulos de rede, protocolos e componentes não essenciais. Um guia de como executar o benchmark reduzido está descrito no apêndice A. Entretanto, surgiu uma limitação crítica: a biblioteca *WolfSSL* não possui suporte nativo às extensões criptográficas *RISC-V* na variante *RV32*. O suporte existente cobre apenas plataformas *RV64*, incluindo acelerações específicas para instruções como *AES*, *SHA-2* e *bitmanip*.

A adaptação desse suporte para *RV32*, entretanto, mostrou-se inviável por três fatores principais:

1. **Complexidade do backend**: o código da *WolfSSL* que integra acelerações para *RISC-V* assume largura de registradores de 64 bits, tanto na lógica de chamadas em *inline assembly* quanto nas estruturas internas. Portá-lo para 32 bits exigiria uma reescrita substancial.

2. **Dependência de intrínsecos inexistentes**: diversas rotinas utilizam funções intrínsecas específicas do compilador para operações criptográficas, que simplesmente não existem na variante *RV32*.

3. **Alto custo de integração**: validar funcionalmente instruções criptográficas completas em algoritmos como AES, SHA-256, CHACHA20 ou POLY1305 exigiria uma infraestrutura de testes e um nível de garantia que extrapolam o escopo deste trabalho.

Assim, o caminho mais viável foi a criação de um *framework* próprio de testes direcionado especificamente às instruções adicionadas, permitindo validação total e controle fino do comportamento observado.

3.2 Implementação do Plugin no VexRiscv

A implementação do suporte em *hardware* para as instruções da extensão *Zbk(b/x/c)* exigiu a criação de um *plugin* dedicado no *VexRiscv*. A arquitetura do processador, construída em *SpinalHDL*, é organizada de forma modular, permitindo adicionar novos blocos de execução, sinais de controle e lógica de decodificação sem alterar o núcleo principal. Essa modularidade foi essencial para isolar corretamente as instruções e preservar a integridade do pipeline original.

```
class BitManipPlugin extends Plugin[VexRiscv] {
        override def setup(pipeline: VexRiscv): Unit = {
        }
        override def build(pipeline: VexRiscv): Unit = {
                execute plug new Area {
                }
                memory plug new Area {
                }
                writeBack plug new Area {
                }
        }
}
```

O primeiro passo consistiu na criação de novos *Stageables*, responsáveis por transportar informações entre os estágios do pipeline. Esses registradores temporários são utilizados para armazenar o opcode específico da instrução, bem como sinais de validade e indicadores de uso de registradores fonte. Em seguida, a etapa de *setup* do plugin foi estruturada para registrar padrões de decodificação associados às instruções implementadas. Essa fase utiliza o serviço padrão de decodificação do *VexRiscv*, permitindo que cada máscara binária seja associada a um conjunto de sinais de controle, como habilitação de escrita no banco de registradores, regras de *bypass* e uso dos operandos *rs1* e *rs2*.

Após a configuração do decodificador, a etapa de *build* do plugin foi responsável por inserir a lógica de execução. Nesse estágio, foram criadas áreas dedicadas dentro do estado **execute**, que implementam a computação combinacional ou sequencial necessária para cada operação. Para instruções simples — como ANDN, ORN ou XNOR — a lógica consiste em expressões diretas em *SpinalHDL*, garantindo baixa latência. Para instruções mais complexas — como REV8, BREV8, ZIP, UNZIP e as operações de multiplicação em campos finitos (CLMUL/CLMULH) — foi necessário introduzir redes combinacionais

mais profundas, operando sobre múltiplos sinais simultaneamente. Outro desafio da escrita do plugin foi manter o equilíbrio entre fidelidade à especificação e viabilidade de síntese. Em *FPGAs* de pequeno porte, certas implementações bit a bit podem gerar profundidades lógicas que afetam a frequência máxima atingível após *place-and-route*. Um trecho com as instruções implementadas pode ser visto a seguir.

```
is(B(OP_ANDN)) { logicRes := rs1 & ~rs2 }
is(B(OP_ORN))  { logicRes := rs1 | ~rs2 }
is(B(OP_XNOR)) { logicRes := rs1 ^ ~rs2 }
is(B(OP_ROL))  { rotateRes := rs1.rotateLeft(shamtRs2) }
is(B(OP_ROR))  { rotateRes := rs1.rotateRight(shamtRs2) }
is(B(OP_RORI)) { rotateRes := rs1.rotateRight(shamtImm) }
is(B(OP_PACK))  { packRes := rs2(15 downto 0) @@ rs1(15 downto 0) }
is(B(OP_PACKH)) { packRes := U(0, 16 bits) @@ rs2(7 downto 0)
                                           @@ rs1(7 downto 0) }
is(B(OP_REV8)) { revRes := rs1(7 downto 0)
                        @@ rs1(15 downto 8)
                        @@ rs1(23 downto 16)
                        @@ rs1(31 downto 24) }
is(B(OP_BREV8)) { revRes := Cat(b3_rev, b2_rev, b1_rev, b0_rev).asUInt }
```

Por fim, após a implementação de todas as instruções da família Zbk(b/c/x), o plugin foi integrado ao *LiteX*, sintetizado na *FPGA* e validado por meio do *framework* de testes desenvolvido. A escrita do plugin representou a etapa central do trabalho, responsável por transformar a especificação teórica das instruções em unidades funcionais prontas para execução real em hardware.

3.3 Construção do Framework de Testes

Para validar e avaliar o conjunto de instruções — tanto as implementadas em hardware quanto aquelas ainda emuladas por software — foi desenvolvido um *framework* de testes massivos integrado ao ambiente *LiteX*. Esse *framework* testa instruções individualmente em escala funcional, garantindo coerência com a especificação da ISA ao comparar os resultados entre a versão em hardware e uma versão equivalente em software puro.

```
        if (passed) {
                printf("[PASS] %-7s | Res: 0x%08" PRIreg " | SW: %5lu | HW:
                ↪  %5lu | Speedup: %lu.%02lux\n", name, hw_res, t_sw, t_hw,
                ↪  int_part, dec_part);
        } else {
                printf("[FAIL] %-7s | Exp: 0x%08" PRIreg " | Got: 0x%08"
                ↪  PRIreg " | SW: %5lu | HW: %5lu\n", name, sw_res, hw_res,
                ↪  t_sw, t_hw);
        }
```

Uma preocupação fundamental no processo de medição foi evitar enviesamento decorrente de otimizações do compilador. Funções em *C* que implementam as versões em software das instruções foram explicitamente marcadas com:

- `__attribute__((noinline))` — evitando que o compilador promova inlining automático.
- `-O0` — desabilitando otimizações para impedir que o compilador substitua ou simplifique algoritmos.

Essas medidas asseguram que o código executado reflita de maneira realista o custo computacional das versões em *software* e em *hardware* das instruções, mantendo a validade estatística dos resultados coletados.

Além disso, trechos grandes de código foram modularizados em seções menores, facilitando tanto a manutenção quanto a instrumentação de cada instrução testada. A seguir são mostrados exemplos de divisão dos blocos principais.

```
uint32_t t0 = get_cycles();
for (int i=0; i<ITERATIONS; i++)
        g_res = FUNC(g_src1, g_src2);
uint32_t t_sw = get_cycles() - t0;
t0 = get_cycles();
for (int i=0; i<ITERATIONS; i++) {
        asm volatile (".option push\n"
        ".option arch, +" #EXT "\n"
        ASM_OP "\n"
        ".option pop"
        : "=r"(g_res)
        : "r"(g_src1), "r"(g_src2));
}
uint32_t t_hw = get_cycles() - t0;
report(NAME, pass, sw_res, hw_res, t_sw, t_hw);
```

Os resultados consolidados são apresentados nas figuras 4 e 5:

Figura 4 – Média de Ciclos: Software vs Hardware

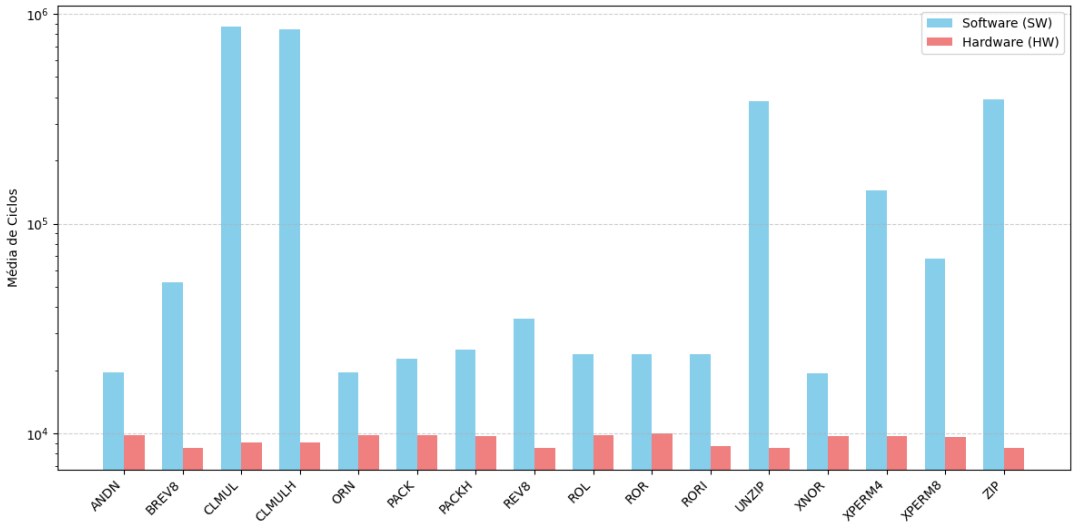
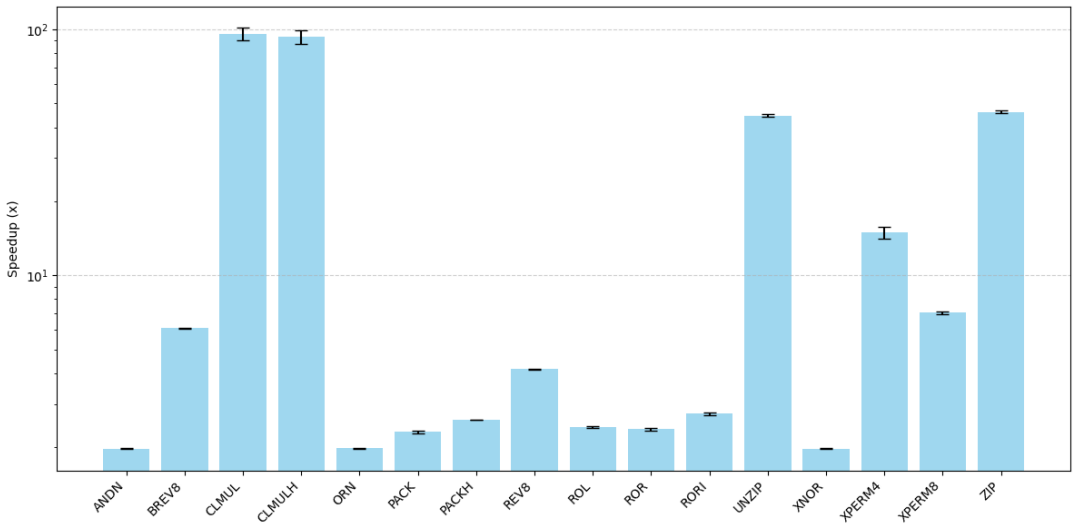


Figura 5 – Speedup Médio com Desvio Padrão por Instrução



Após a implementação completa do plugin de manipulação de bits que implementa o conjunto *Zbk(b/c/x)* no núcleo *VexRiscv* e da instrumentação do *framework* de testes criado para este trabalho, foram conduzidos milhares de iterações para cada instrução, com vetores pseudoaleatórios. Os resultados coletados apresentaram consistência funcional absoluta (todas as operações reportaram `PASS`) e evidenciaram ganhos expressivos de desempenho proporcionados pela execução dedicada em hardware.

Para cada instrução foram extraídas métricas estatísticas agregadas: média dos ciclos em software, média dos ciclos em hardware, *speedups* e desvio padrão dos *speedups* das execuções. As operações lógicas simples (*ANDN*, *ORN*, *XNOR*) apresentaram ganhos modestos, próximos de 2×, como esperado. Já instruções de rotação, reversão e permutação (*ROL*, *ROR*, *REV8*, *XPERM*) exibiram ganhos muito mais elevados. Multiplicações de campo finito (*CLMUL*/*CLMULH*) obtiveram os maiores ganhos observados, reduzindo

rotinas iterativas com dezenas de operações para datapaths combinacionais otimizados. Os dados completos são apresentados na tabela 1.

Instrução	**SW Mean**	**HW Mean**	**Speedup Mean**	**Speedup Std**
ANDN	∼19510	∼9832	**1.98x**	0.0058
ORN	∼19490	∼9809	**1.99x**	0.0029
XNOR	∼19370	∼9755	**1.98x**	0.0029
ROL	∼23790	∼9840	**2.41x**	0.0115
ROR	∼23800	∼10067	**2.36x**	0.0000
RORI	∼23950	∼8792	**2.72x**	0.0058
REV8	∼35350	∼8496	**4.15x**	0.0029
PACK	∼22700	∼9870	**2.31x**	0.0173
PACKH	∼25200	∼9695	**2.59x**	0.0029
BREV8	∼52360	∼8560	**6.10x**	0.0231
ZIP	∼391500	∼8530	**46.00x**	0.5485
UNZIP	∼381000	∼8520	**44.80x**	0.2598
CLMUL	∼848000	∼9060	**93.70x**	5.8312
CLMULH	∼833000	∼9025	**92.00x**	5.8312
XPERM8	∼67700	∼9620	**7.00x**	0.0664
XPERM4	∼148000	∼9680	**15.30x**	0.6351

Tabela 1 – Resultados de desempenho: médias e desvios-padrão do speedup.

3.4 Limitações e Trabalhos Futuros

Apesar dos avanços obtidos no desenvolvimento e na avaliação das instruções de criptografia para núcleos *RISC-V* integrados ao *LiteX* e sintetizados em FPGA, algumas limitações ainda permanecem e abrem espaço para estudos futuros mais aprofundados.

A primeira limitação refere-se ao escopo reduzido das instruções implementadas, que abrange apenas um subconjunto das extensões criptográficas previstas para a arquitetura *RISC-V*. Embora esse conjunto seja suficiente para validar o fluxo de desenvolvimento, sua cobertura funcional limitada impede uma análise mais completa sobre desempenho, latência, área e impactos na microarquitetura. Expansões para incluir instruções adicionais — tanto das extensões padronizadas quanto de variantes experimentais — representam um caminho natural para continuidade do trabalho.

Adicionalmente, os experimentos realizados concentram-se em rotinas de teste estruturadas e em cargas sintéticas geradas pelo *framework* desenvolvido. Embora eficazes para avaliar corretude funcional, esses testes não representam integralmente cenários de uso real, como execução integrada em bibliotecas criptográficas completas ou em protocolos de segurança embarcados. A incorporação dos resultados em pilhas reais, como *TLS*, *SSH* ou algoritmos de PQC, poderia oferecer métricas mais robustas sobre impacto prático.

Por fim, trabalhos futuros também podem investigar otimizações microarquiteturais no núcleo *VexRiscv*, incluindo melhorias em estágio de execução, técnicas de *pipelining*

para instruções criptográficas, integração com unidades funcionais dedicadas e extensões ao conjunto de registradores. Tais aprimoramentos permitiriam analisar ganhos de desempenho sob diferentes configurações de SoCs, ampliando tanto a aplicabilidade quanto a eficiência das soluções criptográficas implementadas.

Em síntese, embora este trabalho estabeleça uma base sólida para a integração e avaliação de instruções criptográficas em arquiteturas *RISC-V*, permanece um amplo conjunto de oportunidades de evolução que podem aprofundar a pesquisa e ampliar o impacto das contribuições apresentadas.

Conclusão

Este trabalho teve como objetivo projetar, implementar e avaliar um conjunto de instruções de manipulação de bit da extensão de criptografia, Zbk(b/c/x) do padrão RISC-V, adicionando suporte por hardware ao processador VexRiscv. A implementação foi acompanhada do desenvolvimento de um framework de testes capaz de estressar intensivamente cada operação, comparando-a com a versão equivalente em software.

Os resultados experimentais confirmaram de forma inequívoca os benefícios da aceleração por hardware. Operações simples, como ANDN/ORN/XNOR, apresentaram ganhos próximos de 2x, enquanto instruções envolvendo permutação complexas, como ZIP/UNZIP, XPERM alcançaram até 45x de aceleração. A maior melhoria foi observada nas instruções de multiplicação de polinômios em campos finitos 2^n — CLMUL e CLMULH — que atingiram speedups superiores a 90x, chegando a ultrapassar 100x em certos vetores.

Além do ganho bruto de desempenho, a integração no pipeline do VexRiscv demonstrou viabilidade técnica: o plugin manteve a compatibilidade com o ecossistema LiteX, não impactou o IPC em operações não relacionadas, trouxe latência mínima adicional ao caminho crítico e manteve correta a lógica de forwarding entre estágios. Do ponto de vista arquitetural, o trabalho evidencia que a adição de instruções especializadas — especialmente em domínios como criptografia, permutação e manipulação de bits — traz benefícios significativos mesmo em microarquiteturas simples como a do VexRiscv, baseada em pipeline escalar.

Essa aceleração é particularmente relevante em sistemas embarcados de baixo custo, nos quais é impraticável depender de núcleos mais robustos ou extensões proprietárias para obter desempenho criptográfico adequado. Em suma, o trabalho alcançou todos os objetivos propostos. Comprovou-se, portanto, que extensões de criptografia no RISC-V são não apenas viáveis, mas extremamente vantajosas quando aplicadas em arquiteturas configuráveis como o VexRiscv, abrindo espaço para SoCs personalizados de alta eficiência energética e desempenho competitivo.

Referências

BACHRACH, J. et al. Chisel: constructing hardware in a scala embedded language. In: **Proceedings of the 49th Annual Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2012. (DAC '12), p. 1216–1225. ISBN 9781450311991. Disponível em: <https://doi.org/10.1145/2228360.2228584>.

BOUTROS, A.; BETZ, V. Fpga architecture: Principles and progression. **IEEE Circuits and Systems Magazine**, v. 21, n. 2, p. 4–29, 2021.

BROWN, S. Fpga architectural research: a survey. **IEEE Design and Test of Computers**, v. 13, n. 4, p. 9–15, 1996.

ENJOY-DIGITAL. **Linux on LiteX-VexRiscv**. 2025. <https://github.com/litex-hub/linux-on-litex-vexriscv>. Acesso em: 30 nov. 2025.

_____. **LiteX: Build your hardware, easily! (GitHub repository)**. 2025. <https://github.com/enjoy-digital/litex>. Acesso em: 28 out. 2025.

FPROX. **RISC-V Scalar Bit Manipulation Extensions**. 2023. <https://fprox.substack.com/p/risc-v-scalar-bit-manipulation-extensions>. Acesso em: 24 out. 2025.

KERMARREC, F. et al. **LiteX: an open-source SoC builder and library based on Migen Python DSL**. 2020. Disponível em: <https://arxiv.org/abs/2005.02506>.

MARSHALL, B.; PAGE, D.; PHAM, T. Implementing the draft risc-v scalar cryptography extensions. In: **Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy**. New York, NY, USA: Association for Computing Machinery, 2021. (HASP '20). ISBN 9781450388986. Disponível em: <https://doi.org/10.1145/3458903.3458904>.

NGUYEN-HOANG, D.-T. et al. Implementation of a 32-bit risc-v processor with cryptography accelerators on fpga and asic. In: **2022 IEEE Ninth International Conference on Communications and Electronics (ICCE)**. [S.l.: s.n.], 2022. p. 219–224.

OPENSSL. **OpenSSL Library**. 2025. <https://openssl-library.org>. Acesso em: 28 out. 2025.

SHAHDAD, M. An overview of vhdl language and technology. In: **23rd ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 1986. p. 320–326.

SPINALHDL. **NaxRiscv: an out-of-order RISC-V core written in SpinalHDL**. 2025. <https://github.com/SpinalHDL/NaxRiscv>. Acesso em: 24 out. 2025.

_____. **SpinalHDL: Scala based HDL (Documentation)**. 2025. <https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Introduction/SpinalHDL.html>. Acesso em: 24 out. 2025.

_____. **SpinalHDL: Scala based HDL (GitHub repository)**. 2025. <https://github.com/SpinalHDL/SpinalHDL>. Acesso em: 24 out. 2025.

_____. **VexRiscv: FPGA friendly 32 bit RISC-V CPU implementation (GitHub repository)**. 2025. <https://github.com/SpinalHDL/VexRiscv>. Acesso em: 28 out. 2025.

TERASIC. **DE10-Standard Development and Education Kit**. 2018. <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1081>. Acesso em: 30 nov. 2025.

THOMAS, D. E.; MOORBY, P. R. **The Verilog Hardware Description Language**. [S.l.]: Springer, 1991.

WATERMAN, A. et al. **The RISC-V Instruction Set Manual Volume I: Unprivileged ISA**. 2025. <https://docs.riscv.org/reference/isa/unpriv/unpriv-index.html>. Acesso em: 24 out. 2025.

_____. **The RISC-V Instruction Set Manual Volume II: Privileged ISA**. 2025. <https://docs.riscv.org/reference/isa/priv/priv-index.html>. Acesso em: 24 out. 2025.

WOLFSSL. **Embedded SSL/TLS Library**. 2025. <https://www.wolfssl.com>. Acesso em: 28 out. 2025.

Apêndices

APÊNDICE A

Execução do benchmark do WolfSSL na Tang Primer 20k

Esse apêndice a seguir é mostrado o passo a passo para executar o benchmark do wolfssl em FPGA, de modo genérico, sem as acelerações e alterações no core RISC-v.

contempla a:

1. Toolchain da Gowin
2. Compiladores cruzados para RISC-v (riscv32/64-unknown-elf-)
3. Toolchain de compilação (Scala/SpinalHDL)

A.1 Toolchain da Gowin

O download da toolchain pode ser encontrado no link:

- <https://cdn.gowinsemi.com.cn/Gowin_V1.9.11.03_Education_Linux.tar.gz>

```
mkdir -p gowin && cd gowin
wget https://cdn.gowinsemi.com.cn/Gowin_V1.9.11.03_Education_Linux.tar.gz
tar -xvf Gowin_V1.9.11.03_Education_Linux.tar.gz
export LD_PRELOAD=/usr/lib64/libfreetype.so.6 # ou
↪  /lib/x86_64-linux-gnu/libfreetype.so
```

A.2 Cross-Compilers (riscv32/64-unknown-elf-)

O download do toolchain gnu pode ser encontrado no link:

- <https://github.com/riscv-collab/riscv-gnu-toolchain/releases>

```
./configure --prefix=/var/opt/riscv --enable-multilib --enable-newlib
↪  --enable-linux --enable-debug-info --with-arch=rv32gc --with-abi=ilp32d
make -j $(nproc) && sudo make install
```

A.3 Toolchain de compilação (Scala/SpinalHDL)

Para geração do VexRiscv, NaxRiscv e outras variantes baseadas em SpinalHDL é preciso do Verilator, sbt e do Java 8+.

```
sudo apt install -y openjdk-8-jdk verilator sbt
```

A.4 Executando a simulação

Agora vamos compilar para executar no simulador Verilog e depois na FPGA:

```
litex_sim --integrated-main-ram-size=0x10000 --cpu-type=vexriscv
↪  --no-compile-gateware
litex_bare_metal_demo --build-path=build/sim/
litex_sim --integrated-main-ram-size=0x10000 --cpu-type=vexriscv
↪  --ram-init=demo.bin
```

Os códigos fontes podem ser compilados na árvore: <https://github.com/enjoy-digital/litex/tree/master/litex/soc/software/demo>

Para encontrar a documentação com os modos de carregamento de programas: <https://github.com/enjoy-digital/litex/wiki/Load-Application-Code-To-CPU>

A.5 Executando na FPGA (Tang Primer 20k)

Simulação com a mesma `sram`: 8192Kb e `ram`: 64Mb da `Tang Primer 20k`:

1. na pasta litex, fazer o download da branch: <https://github.com/arthunix/litex/tree/libwolfssl>

2. na raiz da instalacão do litex criar: <https://github.com/wolfSSL/wolfssl.git> com nome `libwolfssl`.

3. colocar na pasta como `wolfssl/user_settings.h` o arquivo: <https://github.com/arthunix/litex/blob/libwolfssl/litex/soc/software/primer20k_user_settings_template.h>.

```
litex_sim --integrated-sram-size=0x2000 --integrated-main-ram-size=0x4000000
↪  --cpu-type=vexriscv --cpu-variant=full --no-compile-gateware
litex_bare_metal_demo --build-path=build/sipeed_tang_primer_20k/
litex_sim --integrated-sram-size=0x2000 --integrated-main-ram-size=0x4000000
↪  --cpu-type=vexriscv --cpu-variant=full --ram-init=demo.bin
litex_term /dev/ttyUSB1 --kernel=demo.bin
```

```
LiteX minimal demo app built Nov 12 2025 02:51:49
...
test_time_syscalls   - Run Test for Time Syscalls, times, gettimeofday,
↪  current_time
test_memory_syscalls - Test Dynamic Memory Allocation: malloc, calloc,
↪  realloc and free
wolfssl_test         - Run Wolfssl Tests
wolfssl_benchmark    - Run Wolfssl Benchmarks
litex-demo-app> wolfssl_benchmark

--======== Wolfssl Benchmark ========--
Running Wolfssl Init...

Benchmark Test Started
wolfCrypt Benchmark (block bytes 256, min 1.0 sec each)
RNG                      100.0 KiB took 1.058 seconds, 94.513 KiB/s
...

Benchmark Test Completed
Running Wolfssl Cleanup...
```

APÊNDICE B

Port do Litex para DE10-Standard

Este apêndice descreve o processo de adaptação (“port”) do framework LiteX para a placa Terasic DE10-Standard, detalhando as etapas realizadas para disponibilizar os periféricos essenciais, integrar o soft-core e permitir a geração de bitstream, firmware e ambiente de testes para a plataforma.

O LiteX oferece um ecossistema modular para construção de SoCs baseados em FPGA. Entretanto, placas não suportadas oficialmente precisam de:

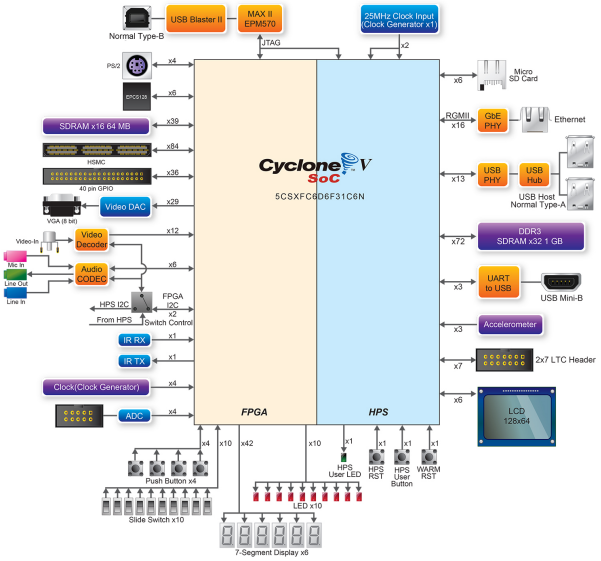
1. Definição da plataforma FPGA no formato LiteX/Migen.
2. Ajustes de pinagem, geradores de clock e interfaces externas.
3. Integração do core de CPU e periféricos desejados.

A placa DE10-Standard, baseada em um Cyclone V 5CSXFC6D6F31C6, não possuía suporte nativo no LiteX, exigindo a criação manual da plataforma e da SoC.

Todos os sinais relevantes foram mapeados para o padrão LiteX de acordo com manual de pinos da placa:

1. Clock de 50 MHz
2. Leds
3. Switches
4. GPIO (40 pinos)
5. SDRAM externa (módulo IS42S16320 do LiteDRAM)
6. VGA

Figura 6 – Diagrama de Blocos da FPGA Altera DE10-Standard



Fonte: Terasic (2018)

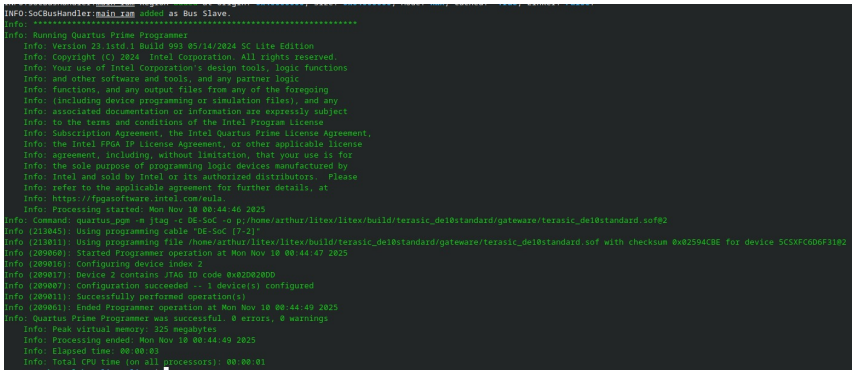
7. UART/I2C

8. Display de 7 segmentos

O port está disponível em: `<https://github.com/arthunix/litex-boards/commit/3f3e2c921193275aaa2b64dd295c7fa54fb6d48b>`.

```
python3 -m litex_boards.targets.terasic_de10standard --cpu-type=vexriscv
↪  --cpu-variant=full --build
litex_bare_metal_demo --build-path=build/terasic_de10standard/
python3 -m litex_boards.targets.terasic_de10standard --cpu-type=vexriscv
↪  --cpu-variant=full --load
```

Figura 7 – Load do VexRiscv na DE10-Standard usando o Litex



APÊNDICE C

Gerar uma CPU Custom do VexRiscv no Litex

Esse apêndice mostra como gerar um core **VexRiscv** personalizado e integrar no **litex** com uma pilha de software.
obs: considerado que já tenha sido feito o **bootstrap** e instalação do **litex**.

```
mkdir litex && cd litex
wget
↪  https://raw.githubusercontent.com/enjoy-digital/litex/master/litex_setup.py
chmod +x litex_setup.py
./litex_setup.py --init --install --user --config full
```

1. dentro da pasta: `cd pythondata-cpu-vexriscv/pythondata_cpu_vexriscv/verilog` corrigir o arquivo `build.sbt`.
2. clonar o repositório: `<https://github.com/SpinalHDL/VexRiscv.git>`.
3. em `lazy val vexRiscv = RootProject(file())` e coloque o caminho da pasta em que o VexRiscv foi clonado.

```
val spinalVersion = "1.12.0"

lazy val root = (project in file(".")).
  settings(
    inThisBuild(List(
      organization := "com.github.spinalhdl",
      scalaVersion := "2.12.18",
      version      := "2.0.0"
    )),
    name := "VexRiscvOnWishbone",
    libraryDependencies ++= Seq(
      "com.github.spinalhdl" %% "spinalhdl-core" % spinalVersion,
      "com.github.spinalhdl" %% "spinalhdl-lib" % spinalVersion,
      compilerPlugin("com.github.spinalhdl" %% "spinalhdl-idsl-plugin" %
      ↪  spinalVersion),
      "org.scalatest" %% "scalatest" % "3.2.17",
      "org.yaml" % "snakeyaml" % "1.8"
    ),
  ).dependsOn(vexRiscv)

lazy val vexRiscv = RootProject(file("/home/arthur/litex/VexRiscv"))
fork := true
```

Depois basta fazer as alterações desejadas e executar a build:

```
sbt compile "runMain vexriscv.GenCoreDefault --csrPluginConfig all
↪  --outputFile VexRiscv_Full"
```

Será gerado o arquivo `VexRiscv_Full.v` ou qualquer variante ou para variante que desejada e vai ser usado na FPGA:

```
sbt compile "runMain vexriscv.GenCoreDefault"
```

```
sbt compile "runMain vexriscv.GenCoreDefault --iCacheSize 0 --dCacheSize 0
↪  --mulDiv false --singleCycleShift false --singleCycleMulDiv false
↪  --bypass false --prediction none --outputFile VexRiscv_Min"
```

```
sbt compile "runMain vexriscv.GenCoreDefault --iCacheSize 2048 --dCacheSize 0
↪  --mulDiv true --singleCycleShift false --singleCycleMulDiv false
↪  --outputFile VexRiscv_Lite"
```

C.1 Integrar a pilha de software litex

Na pasta `litex/soc/software/demo` atualizar o `common.mak` com os `INCLUDES`.

Na pasta `litex/soc/software/demo/demo` atualizar o `Makefile` com os `OBJECTS`.

Caso preciso pode-se criar pastas de bibliotecas com o makefile e incluir na aplicação, basta adicionar o nome `libalgumacoisa` ao arquivo `integration/builder.py`

Simulação com a mesma `sram`: 8192Kb e `ram`: 64Mb da `Tang Primer 20k`:

```
rm -rf demo* build
litex_sim --integrated-sram-size=0x2000 --integrated-main-ram-size=0x4000000
↪  --cpu-type=vexriscv --cpu-variant=full --no-compile-gateware
litex_bare_metal_demo --build-path=build/sim/
litex_sim --integrated-sram-size=0x2000 --integrated-main-ram-size=0x4000000
↪  --cpu-type=vexriscv --cpu-variant=full --ram-init=demo.bin
```

Compilação e execução na FPGA `Tang Primer 20k`:

```
litex_sim --integrated-sram-size=0x2000 --integrated-main-ram-size=0x4000000
↪  --cpu-type=vexriscv --cpu-variant=full --no-compile-gateware
litex_bare_metal_demo --build-path=build/sipeed_tang_primer_20k/
litex_sim --integrated-sram-size=0x2000 --integrated-main-ram-size=0x4000000
↪  --cpu-type=vexriscv --cpu-variant=full --ram-init=demo.bin
litex_term /dev/ttyUSB1 --kernel=demo.bin
```