



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

RISC-V: Ambiente de montagem e simulação

Matheus Y. Matsumoto

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

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Brasília
2017



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

RISC-V: Ambiente de montagem e simulação

Matheus Y. Matsumoto

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

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

Prof. Dr. Donald Knuth Dr. Leslie Lamport
Stanford University Microsoft Research

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia da Computação

Brasília, 26 de março de 2017

Dedicatória

Eu dedico à minha família que sempre me deu suporte, especialmente à minha tia Elizabeth.

Agradecimentos

Agradeço à todos os meus fãs, sem eles nada disso seria possível.

Resumo

O *resumo* é um texto inaugural para quem quer conhecer o trabalho, deve conter uma breve descrição de todo o trabalho (apenas um parágrafo). Portanto, só deve ser escrito após o texto estar pronto. Não é uma coletânea de frases recortadas do trabalho, mas uma apresentação concisa dos pontos relevantes, de modo que o leitor tenha uma ideia completa do que lhe espera. Uma sugestão é que seja composto por quatro pontos: 1) o que está sendo proposto, 2) qual o mérito da proposta, 3) como a proposta foi avaliada/validada, 4) quais as possibilidades para trabalhos futuros. É seguido de (geralmente) três palavras-chave que devem indicar claramente a que se refere o seu trabalho. Por exemplo: *Este trabalho apresenta informações úteis a produção de trabalhos científicos para descrever e exemplificar como utilizar a classe L^AT_EX do Departamento de Ciência da Computação da Universidade de Brasília para gerar documentos. A classe UnB-CIC define um padrão de formato para textos do CIC, facilitando a geração de textos e permitindo que os autores foquem apenas no conteúdo. O formato foi aprovado pelos professores do Departamento e utilizado para gerar este documento. Melhorias futuras incluem manutenção contínua da classe e aprimoramento do texto explicativo.*

Palavras-chave: risc, LaTeX, metodologia científica, trabalho de conclusão de curso

Abstract

O *abstract* é o resumo feito na língua Inglesa. Embora o conteúdo apresentado deva ser o mesmo, este texto não deve ser a tradução literal de cada palavra ou frase do resumo, muito menos feito em um tradutor automático. É uma língua diferente e o texto deveria ser escrito de acordo com suas nuances (aproveite para ler [http://dx.doi.org/10.6061/2Fclinics%2F2014\(03\)01](http://dx.doi.org/10.6061/2Fclinics%2F2014(03)01)). Por exemplo: *This work presents useful information on how to create a scientific text to describe and provide examples of how to use the Computer Science Department's L^AT_EX class. The UnB-CIC class defines a standard format for texts, simplifying the process of generating CIC documents and enabling authors to focus only on content. The standard was approved by the Department's professors and used to create this document. Future work includes continued support for the class and improvements on the explanatory text.*

Keywords: LaTeX, scientific method, thesis

Sumário

1	Introdução	1
1.1	Os Processadores: MIPS, ARM e outros antes do RISC-V	2
1.2	Importância acadêmica e industrial do RISC-V	3
1.3	Ambiente de desenvolvimento	4
1.4	Explicação dos capítulos	4
2	Fundamentação teórica	6
2.1	Arquitetura RISC-V: ISA	6
2.1.1	Objetivos	7
2.1.2	História	8
2.1.3	RISC-V Foundation	8
2.1.4	Open Source	9
2.1.5	Modelo de memória	10
2.1.6	Instruções	10
2.2	Montador	13
2.2.1	Algoritmo de duas passagens	13
2.3	Aplicações web	14
2.3.1	Arquitetura	14
3	Ambiente Proposto	16
3.1	Ambiente proposto	16
3.2	Arquitetura de software	16
3.3	Interface web	17
3.4	Montador	18
3.5	Simulador	19
3.6	Extensibilidade	20
4	Resultados e Avaliação do Sistema	23
4.1	Possíveis erros de montagem	23

4.2 Aritmética Simples	25
4.2.1 Editor de código	28
4.2.2 Simulação	28
4.3 Sequência de Fibonacci	29
4.3.1 Editor de código	29
4.3.2 Simulação	31
5 Conclusões	33
5.1 Objetivos atingidos	33
5.2 Pontos positivos e negativos	33
5.3 Dificuldades	34
5.4 Melhorias e trabalhos futuros	34
Referências	36

Lista de Figuras

2.1	Formatos de instruções da ISA básica	10
2.2	Encoding dos imediatos de cada tipo de instrução	11
2.3	Ilustração da arquitetura Cliente-Servidor	14
3.1	Página inicial, mostra o editor de texto com um código exemplo.	18
3.2	Resultados da montagem em binário.	19
3.3	Resultados da montagem em hexadecimal.	20
3.4	Resultados da montagem em formato MIF para FPGA.	21
3.5	Resultados do simulador. Código montado, mapa da memória, registrado- res.	21
3.6	Resultados do simulador. Console de saída de informações do sistema. . . .	22
4.1	Erro de declaração múltipla de símbolo.	24
4.2	Erro de declaração de diretiva string.	24
4.3	Erro de imediato para o tamanho 32 de palavra 32 bits.	25
4.4	Erro de operando inválida, operando de tipo diferente do esperado.	25
4.5	Erro de operação inválida, instrução não existente na arquitetura imple- mentada.	26
4.6	Erro de operação inválida, diretiva não existente na arquitetura implemen- tada.	26
4.7	Erro de símbolo inválido, símbolo não foi declarado.	27
4.8	Erro de argumentos de diretiva inválido, o tipo ou número de argumentos está incorreto.	27
4.9	Demonstração de operações matemáticas básicas.	28
4.10	Demonstração de operações matemáticas básicas.	28
4.11	Primeira parte do código que gera sequência de fibonacci.	29
4.12	Segunda parte do código que gera sequência de fibonacci.	30
4.13	Primeira parte dos resultados gerados da sequência de fibonacci.	31
4.14	Segunda parte dos resultados gerados da sequência de fibonacci.	32

Capítulo 1

Introdução

Este projeto é um estudo da arquitetura RISC-V, e implementações de seus conceitos em ferramentas auxiliares. Estas ferramentas serão utilizadas para pesquisa, ensino, e aplicações de conceito estudados na área de arquitetura de computadores utilizando a arquitetura mencionada.

RISC-V é uma arquitetura de conjunto de instruções aberta, criada na Universidade da Califórnia, em Berkeley. Originalmente foi pensada para ser utilizada na pesquisa e ensino da área de arquitetura de computadores, mas está se tornando um padrão de arquitetura aberta para a indústria. [1]

Seu nome é pronunciado na língua inglesa como "*risc five*". O motivo de ser "*five*" é devido ao fato de que é o quinto maior projeto de uma ISA RISC desenvolvida na UC Berkeley. As primeiras foram RISC-I, RISC-II, SOAR, e SPUR. O numeral romano "V" de RISC-V também funciona com significado de "*variations*" e "*vectors*".

Ao iniciar o projeto haviam poucas ferramentas relacionada a arquitetura RISC-V, e as ferramentas que haviam eram de difícil instalação, e configuração. Para este projeto foi implementado um montador, e um simulador para a ISA base RV32I. Este módulo base da arquitetura cobre os pontos principais de uma arquitetura funcional.

O sistema foi desenvolvido com intuito de ser multiplataforma, extensível, fácil utilização, e também para que outros possam facilmente continuar a evolução do projeto. Utilizar a plataforma Web para a disponibilização facilita a utilização imediata em qualquer dispositivo com um Browser e acesso a internet, sem a necessidade de instalações e configurações iniciais que podem trazer dificuldades e desviar a atenção principal que é o estudo da arquitetura RISC-V.

Estudar arquitetura de computadores é uma tarefa difícil, quanto mais fáceis e didáticas forem as ferramentas utilizadas, melhor para o aprendizado. Um grande ecossistema de uma solução pode fazer uma tecnologia evoluir muito mais rápido, por isso é muito

importante desenvolver ferramentas que auxiliam a aprendizagem e utilização dessa nova arquitetura.

1.1 Os Processadores: MIPS, ARM e outros antes do RISC-V

As ISAs MIPS e ARM tiveram grande influência na arquitetura RISC-V, porém existem alguns detalhes técnicos que desfavorecem o uso destas arquiteturas ao invés de criar a nova RISC-V, no ponto de vista dos autores do RISC-V [2],

O MIPS é uma ISA criada no começo dos anos 80, em Stanford, seu design utiliza a filosofia RISC, e facilita implementação de pipelines. MIPS foi implementado comercialmente pela primeira vez no processador R2000, em 1986.

Algumas desvantagens que desencorajam o uso do MIPS principalmente para implementações de alta performance:

- A ISA é exageradamente otimizada para o padrão de pipeline de cinco estágios em ordem, como jumps e branches são atrasados, isso complica implementações super-escalares e super-pipelines. Sendo que o recurso de branch delay slot, não pode ser retirada por questões de compatibilidade.
- A ISA provê um pobre suporte para código de posições independentes. A revisão de 2014 melhorou endereçamento relativo ao contador de programa, porém as chamadas ainda necessitam geralmente de mais de uma instrução.
- Imediatos de 16 bits consomem muito espaço de codificação de instruções, deixando pouco espaço para futuras extensões da ISA, ou trabalhar com instruções comprimidas.
- Multiplicações e divisões utilizam recursos especiais de arquitetura.

Além dessas e algumas outras questões técnicas, MIPS não pode ser utilizado em várias situações pelo fato de ser proprietária. Historicamente, a patente da MIPS Technologies sobre instruções de *load* e *store* desalinhados, preveniu terceiros de implementar totalmente sua ISA.

Outra arquitetura popular que teve influência no MIPS é a arquitetura ARM, mais especificamente as arquiteturas ARMv7 e ARMv8. Estas arquitetura são baseadas na filosofia RISC, e são de longe as mais utilizadas no mundo. A arquitetura é desenvolvida hoje pela empresa britânica ARM Holdings. Inicialmente criada pela Acorn Computers Limited de Cambridge, Inglaterra, entre 1983 e 1985, baseado no processador RISC-I da Berkeley.

A utilização do ARM foi desconsiderada principalmente pelos seguintes motivos,

- Quando o projeto foi iniciado, a arquitetura estava na versão quatro. Nesta versão ainda não havia suporte para 64 bits.
- A ISA possui embutida uma ISA para instruções comprimidas e uma ISA de tamanho variável, porém são codificadas de forma diferente da ISA comum 32 bits, portanto os decodificadores são ineficientes, energeticamente, temporalmente e em relação a custo.
- A ISA tem muitos recursos que complicam implementações. Por exemplo o contador de programa é um dos registradores endereçáveis, e o bit menos significativo do contador seleciona qual ISA está executando (ARM tradicional ou de instruções comprimidas) e assim a instrução ADD, por exemplo, consegue modificar o valor do contador de programa.
- Mesmo que fosse possível implementar a arquitetura ARM legalmente, a quantidade de instruções é gigantesca e seria tecnicamente bem desafiador.

Entre outros motivos, estes são alguns dos principais que levaram os autores do RISC-V desenvolverem sua própria ISA e não utilizar algumas já consolidadas para seus projetos.

Além dessas, existem outras que foram consideradas, como *SPARC*, arquitetura open-source desenvolvida pela Sun Microsystems, *Alpha*, desenvolvida pela Digital Equipment Corporation, *OpenRISC*, evolução da arquitetura educacional open-source *DLX* desenvolvida pelo Patterson e Hennessy.

1.2 Importância acadêmica e industrial do RISC-V

RISC-V foi desenvolvido por Krste Asanović, Andrew Waterman, e Yunsup Lee na Universidade da Califórnia, Berkeley, com colaboração de David Patterson. Eles precisavam de uma ISA que pudessem estudar, implementar com liberdade para fins acadêmicos, porém como visto na seção anterior, nenhuma se encaixava perfeitamente no que eles precisavam. Na maioria das vezes, muito complexas, ou muito específicas, ou simplesmente fechadas.

Com essa necessidade surgiu o RISC-V, uma arquitetura de conjunto de instruções de propósito geral, open-source, modular, com a ambição de ser um conjunto universal, livre e grátis para utilização em um amplo espectro de problemas, desde soluções embarcadas, até soluções de alta performance, e aprendizagem de máquina por exemplo.

Com grandes empresas colaboradoras como Google, Nvidia, Western Digital, Oracle, e também as companhias de chip IBM, AMD, e Qualcomm, a ISA tem ganhado espaço nas áreas comerciais também.

A Western Digital, por exemplo, anunciou em um Workshop que irá liderar a indústria na troca por mais ISAs RISC-V utilizando um bilhão de núcleos RISC-V em seus dispositivos [3]. Em outro workshop, NVIDIA apresentou por que e como irá implementar novos núcleos para seus micro-controladores utilizando o RISC-V. [4]

Os criadores do RISC-V fundaram uma empresa chamada SiFive, na qual eles vendem kits de desenvolvimento estilo arduino com processadores RISC-V, ou então kits mais avançados capazes de rodar linux, como raspberry pi, entre outras coisas.

Os ataques Spectre e Meltdown que exploram vulnerabilidades na arquitetura de processadores modernos, mostram uma grande importância tanto acadêmica e industrial. Pois a mitigação desses ataques não podem ser estudados e resolvidos com facilidade em arquiteturas fechadas como as da Intel ou ARM. Porém utilizando ISAs open-source podemos estudar melhores alternativas de como resolver esse tipo de problemas arquiteturais com muito mais facilidade e rapidez.

1.3 Ambiente de desenvolvimento

Neste projeto foi desenvolvido um ambiente de desenvolvimento para a arquitetura RISC-V. O ambiente inclui um editor de texto para a linguagem assembly, um montador e um simulador. A implementação foi realizada na plataforma web, maiores detalhes serão apresentados no capítulo 3.

Este conjunto de ferramentas auxiliam o processo de aprendizagem da arquitetura. O fato da plataforma ser acessível por qualquer navegador torna a imersão inicial mais eficaz, pois a pessoa interessada não precisará gastar tempo e esforço com problemas que podem ocorrer na instalação ou configuração da ferramenta.

1.4 Explicação dos capítulos

Este primeiro capítulo apresenta o contexto, motivação, objetivo do projeto que foi realizado.

No capítulo 2, apresentaremos a fundamentação teórica para o desenvolvimento do projeto, incluindo maiores detalhes técnicos da arquitetura e conhecimentos necessários para entender o desenvolvimento do ambiente proposto.

No capítulo 3 mostraremos a implementação, arquitetura de software, decisões de projeto, e as aplicações da fundamentação teórica no projeto.

No quarto capítulo apresentaremos o resultados que obtivemos, exemplos de utilização com uma sequência lógica.

Quinto e último capítulo descreve objetivos atingidos, pontos positivos e negativos, dificuldades e possíveis melhorias junto com idéias de implementações futuras para melhorar o projeto.

Capítulo 2

Fundamentação teórica

Este capítulo aborda a base teórica necessária para o desenvolvimento do projeto.

2.1 Arquitetura RISC-V: ISA

Sua arquitetura obedece aos padrões RISC (Reduced Instruction Set Computing), tendo instruções simples e completas. Foi projetada para ser rápida, ocupar pouco espaço físico, ter baixo consumo de energia, ser extensível, e compatível com entre suas versões. Por ser reduzida, se encaixa perfeitamente para fins acadêmicos, e pesquisas.

Outra característica importante é a sua extensibilidade. Por padrão sua base é inteira, para arquiteturas de 32, 64 e 128 bits, porém existem módulos de extensão. Para descrever quais implementações são utilizadas utilizam-se as nomenclaturas RV32I, RV64I e RV128I, para as implementações padrões RISC-V 32 bits, 64, e 128.

Existem módulos padrões e não-padrões de extensões [1]:

- Os módulos padrões são aqueles que não possuem conflitos entre si e é utilizado para propósitos gerais.
- Os módulos não-padrões são módulos especializados, podendo conflitar com outros módulos. A previsão é de que no futuro haja muitos módulos desse tipo.

As padrões desenvolvidas atualmente adicionam as letras "MAFDQLCBJTPVN", sendo que cada letra representa uma extensão

- M: Multiply/Divide
- A: Atomic
- F: Single-Precision Floating-Point
- D: Double-Precision Floating-Point

- Q: Quad-Precision Floating-Point
- L: Decimal Floating-Point
- C: 16-bit Compressed Instructions
- B: Bit Manipulation
- J: Dynamic Languages
- T: Transactional Memory
- P: Packed-SIMD Extensions
- V: Vector Extensions
- N: User-Level Interrupts

As extensões IMAFD são chamadas de extensões de propósito geral e são abreviadas por G, por exemplo, uma arquitetura de 32 bits que utilizam todas as extensões de propósito geral é chamada de RV32G.

Ainda existe uma outra variação que é a extensão "E", que se difere das outras pois é projetada para sistemas embarcados. Este módulo diminui a quantidade de registradores para 16 e o tamanho também é reduzido para 16 bits.

2.1.1 Objetivos

Seus projetistas sempre são perguntados o motivo ao qual eles quiseram desenvolver uma nova ISA. Alguns dos motivos para o qual usar uma ISA comercial são a existência de suporte de um ecossistema de software, tanto ferramentas de desenvolvimento, portabilidade e ferramentas educacionais, outros benefícios seriam a grande quantidade de documentação, tutoriais e exemplos para o desenvolvimento.

Porém estas vantagens são pequenas na prática, e listam várias desvantagens ao utilizar ISAs comerciais,

- ISAs comerciais são proprietárias
- ISAs comerciais são populares somente em alguns nichos do mercado
- ISAs comerciais vem e vão
- ISAs populares são complexas
- ISAs comerciais dependem de outros fatores para trazer aplicações
- ISAs comerciais populares não são projetadas para extensibilidade

- Uma ISA comercial modificada é uma nova ISA

Na opinião dos projetistas do RISC-V, em um sistema computacional, a ISA é a interface mais importante, e não existe razão pra que esta seja proprietária [2]. E uma ISA livre e aberta tem um potencial de inovação, redução de custos muito maior.

A ISA RISC-V tem o propósito de uso geral, ou seja, você pode implementar esta arquitetura para uma variedade de problemas, desde Internet das coisas até aplicações de alta performance. Para isso os projetistas tiveram a preocupação de fornecer uma ISA o mais simples possível e modular, portanto para resolver tipos diferentes de problemas se pode utilizar módulos diferentes já citados anteriormente neste capítulo.

2.1.2 História

A ISA RISC-V foi originalmente desenvolvida na Universidade da Califórnia, Berkeley, na Divisão de Ciência da computação, no departamento de Engenharia Elétrica e Ciência da Computação. Baseada na experiência com projetos passados de seus projetistas, a definição da ISA foi iniciada no verão de 2010.

Em 13 de maio de 2011, foi lançada a primeira documentação para nível de usuário, *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA* [5]

Em 6 de maio de 2014, saiu a versão 2.0 do manual, *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA, Version 2.0* [6]

Em 7 de maio de 2017, a versão 2.2 que utilizamos neste projeto foi lançada, *The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2* [1].

Os primeiros processadores RISC-V fabricados foram escritos em Verilog e manufaturados em tecnologia de pré-produção de 28 nm FD-SOI (*Fully Depleted Silicon On Insulator*) da companhia STMicroelectronics com o nome *Raven-1* em maio de 2011.

Sendo as ultimas implementações fabricas documentadas no manual os processadores EOS22 em março de 2014, pela IBM usando a tecnologia 45nm SOI.

A ISA tem sido utilizada em cursos de Universidade da Califórnia em Berkeley desde 2011.

2.1.3 RISC-V Foundation

A *RISC-V Foundation* é uma organização sem fins lucrativos, criada para direcionar futuro desenvolvimento e incentivar a utilização da ISA RISC-V. [7]

O presidente do conselho atualmente é Krste Asanovic, professor do departamento de Engenharia elétrica e ciência da computação na Univerisdade da Califórnia em Berkeley. Também co-fundado da empresa SiFive Inc., a qual incentiva do uso comercial de processadores RISC-V.

E o vice-presidente é o professor David Patterson, muito conhecido pelo livro *Computer Architecture: A Quantitative Approach*, que escreveu juntamente com John Hennessy, e suas pesquisas relacionadas a RISC, RAID, e Redes de estações de trabalho.

Outros membros incluem:

- Zvonimir Bandic, pesquisador e diretor da Western Digital Corporation.
- Charlie Hauck, CEO da Bluespec Inc.
- Frans Sijstermans, vice presidente de engenharia da NVIDIA.
- Ted Speers, chefe de arquitetura de produtos e planejamento do grupo SoC da Microsemi.
- Rob Oshana, gerente de desenvolvimento de software e negócios em segurança na NXP Semiconductors.

e também, Sue Leininger, gerente de comunidade e Rick O'Connor, director executivo. [8]

2.1.4 Open Source

O modelo de licenciamento que RISC-V utiliza é a *BSD Open Source License*. Ou seja, em caso de utilização, apenas dar créditos aos autores, no caso a UC Berkeley. [9]

O fato da ISA ser Open Source traz grandes vantagens, principalmente na parte de distribuição e compartilhamento.

Na parte comercial por exemplo, qualquer pessoa pode criar suas implementações para seus objetivos específicos e comercializar, com seu código fonte sendo aberto ou fechado, apenas é requisitado pela licença que os autores sejam reconhecidos. Diminuindo custos de uso de patentes ou implementações do zero.

Outra vantagem, defendida pelos autores, também defendida no mundo do software livre é a questão de vulnerabilidades na solução. Apesar de parecer contra intuitivo, o fato do código ser aberto, as vulnerabilidades podem ser encontradas e consertadas com maior rapidez, sendo dispensável um auditor para lidar com vulnerabilidades implantadas por desenvolvedores maliciosos de dentro da própria empresa.

Ou mesmo que as vulnerabilidades não tenham sido feitas de forma maliciosa, bugs acontecem e o fato do código ser fechado pode deixar que esta fique escondida por algum tempo antes de poder ser descoberta e explorada, como foi o caso das vulnerabilidades *Spectre* e *Meltdown* [10] que ganharam atenção na mídia recentemente e estão diretamente ligados ao mundo dos processadores por explorarem vulnerabilidades arquiteturais [11].

A RISC-V Foundation escreveu em seu site oficial que não foram encontrados impactos em nenhuma implementação até janeiro de 2018. Apesar dos ataques não serem específicos

de uma ISA ou outra, mostra uma vantagem de se ter uma ISA aberta, a velocidade de se corrigir os erros pela comunidade é muito maior. E a possibilidade de poder experimentar e testar novas formas eficientes para resolver estes problemas é histórica. [12].

2.1.5 Modelo de memória

O espaço de endereços do RISC-V é endereçado byte a byte, na convenção de ordenação little-endian. Diferente de arquiteturas como x86 ou ARM, RISC-V utiliza somente endereçamento base+offset com imediato de 12 bits. Pela especificação todos os loads e stores desalinhados, porém como é feito depende da implementação. A ISA contém a instrução FENCE para ordenação explícita em threads e outras sincronizações.

2.1.6 Instruções

A ISA básica, ou seja o RV32I, possui quarenta e sete instruções, sendo possível reduzir para trinta e oito com implementações mais simples, descartando algumas instruções de chamadas de sistemas, e de controle de estado de registradores, ou sincronização de memória para recurso de threading, substituindo por uma instrução SYSTEM geral, ou substituindo por NOPs.

Existem seis tipos de instruções básicas, como mostradas na figura 2.1,

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
funct7				rs2			rs1		funct3		rd			opcode		R-type		
imm[11:0]						rs1		funct3		rd			opcode		I-type			
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]										rd				opcode		U-type		
imm[20]		imm[10:1]				imm[11]		imm[19:12]				rd			opcode		J-type	

Figura 2.1: Formatos de instruções da ISA básica

Fonte: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2 [1]

sendo as instruções do tipo B uma variação do tipo S, e as instruções do tipo J uma variação do tipo U. Algumas convenções adotadas são, as posições dos registradores *source* rs1, e rs2, e o registrador de destino rd, sempre ocupam o mesmo lugar nos diferentes formatos, pois isso simplifica a decodificação, além disso, todos os imediatos são estendidos pelo sinal, exceto pelos imediatos das instruções de CSR (*"Control and Status Registers"*).

A figura 2.2, mostra os immediatos produzidos por cada tipo de intrução e de qual parte da intrução vem cada bit do imediato.

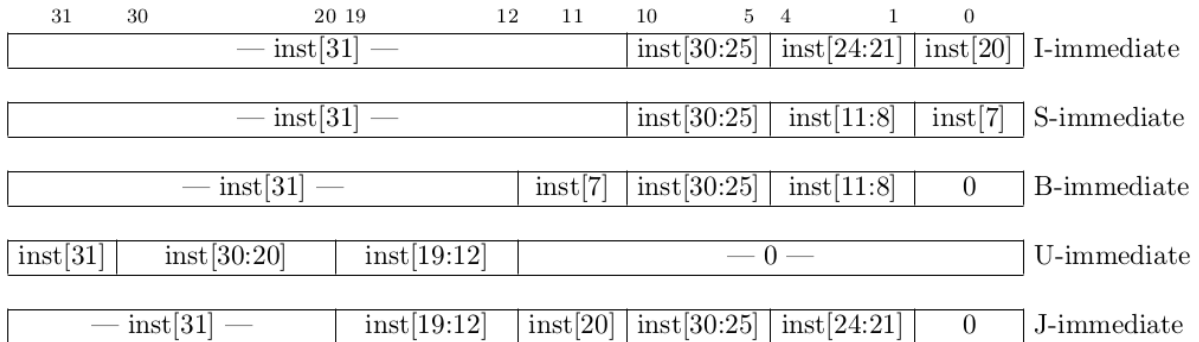


Figura 2.2: Encoding dos immediatos de cada tipo de intrução

Fonte: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2 [1]

As intruçõs do tipo R, sãs as do tipo registrador-registrador, utilizam trẽs registradores como argumentos, dois sendo fonte e um de destino por exemplo:

- ADD x1, x2, x3. Soma os valores do conteúdos nos registradores x2 e x3 e armazena em x1.
- SUB x1, x2, x3. Subtrai os valores do conteúdos nos registradores x2 e x3 e armazena em x1.
- XOR x1, x2, x3. Realiza a operação binária XOR dos conteúdos dos registradores x2 e x3 e armazena o resultado em x1.
- SLT x1, x2, x3. Set Less Than. Insere o valor 1 no registrador x1, se o conteúdo de x2 for menor que o conteúdo de x3, considerando o sinal. Caso contrário, insere o valor 0 em x1.
- SLL x1, x2, x3. Shift Left Logical. Armazena em x1 o valor de x2 deslocado para esquerda do valor dos cinco bits mais baixos de x3.

Outras intruçõs incluem, AND, OR, SLTU, SRL, SRA...

As do tipo I, sãs as do tipo registrador-imediato, sãs muito parecidas com as do tipo R porém utilizam um valor explícito do código ao invés de um registrador como um dos argumentos, exemplos:

- ADDI x1, x2, imm. Soma o valor contido no registrador x2 e imm e armazena em x1.

- SUBI x1, x2, imm. Subtrai o valor de imm do conteúdo do registradores x2 e armazena em x1.
- XORI x1, x2, imm. Realiza a operação binária XOR dos conteúdos dos registradores x2 e imm e armazena o resultado em x1.
- SLTI x1, x2, imm. Set Less Than Immediate. Insere o valor 1 no registrador x1, se o conteúdo de x2 for menor que o valor imm, considerando o sinal. Caso contrário, insere o valor 0 em x1.
- SLLI x1, x2, imm. Shift Left Logical Immediate. Armazena em x1 o valor de x2 deslocado para esquerda do valor imm.

Outras instruções incluem, ANDI, ORI, SLTUI, SRLI, SRAI, e também as instruções de LOAD, e JALR (*"Jump And Link to Register"*).

As instruções do tipo S são dedicadas as operações de STORE, ou seja, armazenar informações na memória,

- SB x1, x2, imm. Store Byte. Armazena o byte menos significativo contido em x2 no endereço que se dá pelo conteúdo de x1 mais o imm
- SH x1, x2, imm. Store Halfword. Armazena a halfword menos significativa contida em x2 no endereço que se dá pelo conteúdo de x1 mais o imm
- SW x1, x2, imm. Store Word. Armazena a word contida em x2 no endereço que se dá pelo conteúdo de x1 mais o imm.

As instruções do tipo B são utilizadas nas operações de BRANCH, ou seja, saltos condicionais,

- BEQ x1, x2, label, Branch Equal. Se os registradores x1 e x2 tiverem o mesmo conteúdo, o salto é realizado
- BLT x1, x2, label, Branch Less Than. Se o valor de x1 for menor que o de x2, o salto é realizado
- BLTU x1, x2, label, Branch Less Than Unsigned. Se o valor inteiro sem sinal de x1 for menor que x2 o salto é realizado

As instruções do tipo U são utilizadas nas operações LUI e AUIPC,

- Load Upper Immediate, LUI x1, imm. Carrega os 20 bits de imediato nos 20 bits mais significativos de x1 e completa com zeros
- Add Upper Immediate to PC, AUIPC. Adiciona 12 bits zerados aos 20 bits do imediato e soma ao valor do contador de programa.

E finalmente a instrução tipo J para saltos incondicionais,

- JAL x1, addr. Jump And Link. Salva o valor de PC+4 em x1 e salta para o valor de PC+addr

2.2 Montador

Um montador é basicamente um software que traduz um programa-fonte em linguagem de máquina. O montador traduz códigos da linguagem *Assembly*, e é um passo intermediário entre a compilação de um código em linguagem de alto nível como *C/C++*, *Java*, *Fortran* para linguagem de máquina.

2.2.1 Algoritmo de duas passagens

O algoritmo utilizado para a implementação do montador se chama Algoritmo de duas passagens, esse nome é devido ao fato de que dado o código fonte da aplicação precisa ser lido pelo algoritmo duas vezes para poder realizar a montagem completa. É o algoritmo mais simples de ser implementado e tem grande eficiência. O algoritmo de uma passagem é mais rápido porém consome mais memória, além de outros detalhes.

O processo de montagem pode ser dividido em quatro partes dentro do algoritmo,

- Análise Léxica, esta parte do código é responsável por limpar o código o máximo possível, retirando todo tipo de informação desnecessária como por exemplo, espaços em branco, tabulações, e comentários. No processo é criado grupo de tokens.
- Análise Sintática, também conhecido como parser em inglês, faz uma checagem de algumas regras da linguagem, por exemplo uma instrução inexistente sendo utilizada, ou uma diretiva, ou símbolos em lugares indevidos, por exemplo, se ao invés de utilizar vírgulas entre os argumentos de uma instrução, utilizar asteriscos.
- Análise Semântica verifica erros utilizam sintaxes corretas, porém em lugares errados, por exemplo ao tentar utilizar uma instrução ADD com um imediato como argumento, ou então tentar realizar um JUMP para uma label não declarada.
- Geração de código é a parte mais direta, com o auxílio de uma tabela da linguagem, transforma os tokens em código binário com a codificação necessária para o processador saber onde estão as informações necessárias.

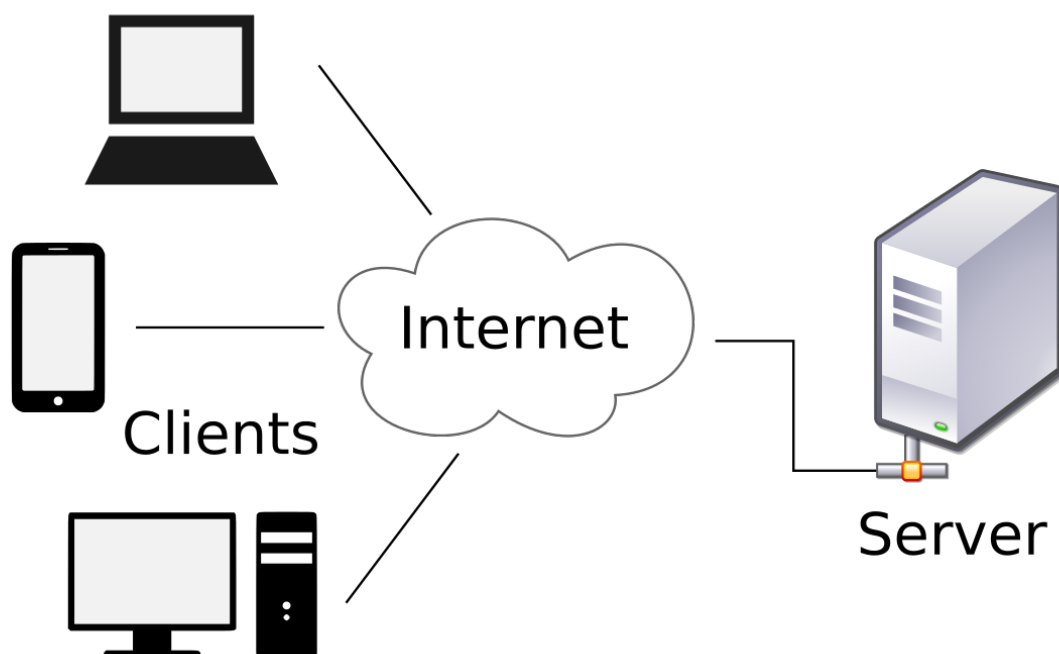


Figura 2.3: Ilustração da arquitetura Cliente-Servidor

Fonte: <https://upload.wikimedia.org/wikipedia/commons/c/c9/Client-server-model.svg>

2.3 Aplicações web

As aplicações web, são aquelas projetadas para que sua utilização seja feita através de um navegador com acesso à Internet.

2.3.1 Arquitetura

Como a maioria das aplicações web, utilizamos o modelo Cliente - Servidor, ilustrado na figura 2.3.

Vantagens e desvantagens

A grande vantagem e motivo principal pela escolha dessa plataforma é poder ser utilizado de qualquer dispositivo com um browser moderno e acesso à internet. Qualquer pessoa pode acessar a solução acessando um link e começar a desenvolver e estudar códigos escritos para a arquitetura RISC-V. Outra vantagem é a correção de bugs e atualizações para novas versões. Para que os usuários tenham seus softwares atualizados basta atualizar o software em um ponto apenas.

Uma desvantagem é ter o custo de um servidor rodando a aplicação para que seja acessível por vários usuários. Outro fator crítico é ter um único ponto de falha, diferente

de uma rede peer-to-peer distribuída. Em relação as desvantagens elas não representam uma significância alta pois se trata de uma ferramenta livre, ou seja, caso haja algum tipo de indisponibilidade do servidor, o usuário poderá baixar o software e rodar em sua própria máquina.

O *frontend* e o *backend*

Podemos dividir aplicações em duas partes, o *frontend* e o *backend*, sendo o primeiro, a parte visual da aplicação, no caso deste projeto em específico, a parte que irá ser renderizada no browser do usuário, HTML, CSS e JavaScripts. O segundo é o sistema com suas regras de negócio, gerências, armazenamento e recuperação de dados, entre outros.

API

API é uma sigla para *Application Program Interface*, é uma interface de comunicação entre aplicações. Um API é definida por um conjunto de funcionalidades pré-implementadas para envio e recebimento de dados. Neste projeto utilizamos uma WEB API, com isso um servidor fornece seus serviços através de chamadas HTTP através de URLs.

A interação entre o *frontend* e o *backend* é realizada por chamadas à API. Basicamente, quando um usuário clica em algum elemento interativo da aplicação, este clique envia dados ao servidor em formato *JSON* e o servidor irá retornar dados no mesmo formato.

Dessa maneira, o *frontend* e o *backend* funcionam de forma independente, e caso seja necessário criar um novo *frontend*, por exemplo para um dispositivo diferente, ou então uma versão desktop ao invés de utilizar o browser, é possível continuar utilizando o mesmo *backend*.

Capítulo 3

Ambiente Proposto

Este capítulo descreve com detalhes o ambiente de desenvolvimento que foi realizado neste projeto.

3.1 Ambiente proposto

O desenvolvimento do ambiente foi realizado para a plataforma web. Consiste em três partes principais:

- Editor de texto
- Montador
- Simulador

Toda a parte de interação com o usuário utiliza as tecnologias web: *javascript*, para ter uma interface dinâmica, utilizando a biblioteca *jQuery*. Para a estruturação utilizou-se a linguagem de marcação *HTML*, e a folha de estilos *CSS* para formatação do layout.

Apesar de ter sido desenvolvido para rodar na web, os componentes "Montador" e "Simulador", podem ser utilizados separados, por exemplo em linha de comando. Basta ter instalado um interpretador *Python 3.x*

3.2 Arquitetura de software

Como na maioria das aplicações web, utilizamos o modelo Cliente-Servidor, como ilustrado anteriormente na figura 2.3. Neste modelo, o usuário é um cliente, que faz requisições ao servidor através da internet.

Usuário faz requisição da página inicial, onde ele pode escrever seu código, então a partir disso ele irá enviar este código para o servidor, requisitando a montagem, e então receberá a saída do montador, ou a mensagem apropriada no caso de erros.

Com o código montado, pode se salvar em arquivos os dados da montagem em diferentes formatos, binário, hexadecimal ou em formato *MIF* ("*Memory Initialization File*"), utilizado para inicializar memórias em FPGA's da Altera. Esses dados podem ser enviados novamente para o servidor em uma nova requisição de simulação, e então o servidor irá responder os resultados do programa, qual o estado de memória final, valores dos registradores, e se houver, mensagens de saída.

Para melhorar a interação do usuário com a ferramenta, foi utilizado a metodologia de *Single Page Application* [13]. Desta maneira, utilizamos apenas uma página e carregamos conteúdos dinamicamente na mesma página.

Para a implementação da metodologia foi utilizado a biblioteca *jQuery*. Existem tecnologias mais modernas mais apropriadas para a implementação de uma SPA, porém foi decidido não utilizá-las por questões de tempo para a aprendizagem destas.

O código *jQuery* faz chamadas HTTP assíncronas ao backend usando os métodos POST e GET ao pressionar algum botão de ação do sistema.

3.3 Interface web

A interface web foi feita com as tecnologias padrões da web, HTML, CSS e Javascript. E faz parte do frontend da aplicação, onde o usuário irá interagir com a ferramenta.

Para a parte dinâmica do site foi utilizada a biblioteca *jQuery*, facilitando o desenvolvimento de funcionalidades no front-end principalmente na manipulação de elementos da tela.

Para o layout foi utilizado o framework *Materialize CSS*, com sua utilização perdemos menos tempo em detalhes de layout, e podemos focar mais nas funcionalidades.

Na figura 3.1, vemos a tela inicial do sistema, a parte do editor de texto. No lado esquerdo da tela está o menu global, que está presente em todas as telas e serve para a navegação principal do sistema.

Ao lado direito está o conteúdo da tela que consiste no editor de texto e um console de saída. O editor de texto utiliza a biblioteca open-source *CodeMirror* [14], escrita em javascript para criar editores de texto baseados em HTML.

Após escrever o seu código, o usuário clica no botão *ASSEMBLE*, se o montador não retornar nenhum erro não haverá mensagens no console e o botão *RUN* será habilitado. Ao apertar o botão *RUN* a tela irá ser trocada para a tela de simulação, que mostraremos nas próximas seções.



Figura 3.1: Página inicial, mostra o editor de texto com um código exemplo.

3.4 Montador

Utilizou-se neste projeto o algoritmo de duas passagens para montagem. Implementamos apenas as funcionalidades básicas para traduzir códigos *assembly RISC-V* para código de máquina.

O montador lida diretamente com a entrada do usuário, por isso essa parte pode ser considerada a mais crítica no projeto inteiro. Para que o sistema funcione corretamente a entrada do usuário, ou seja o código fonte, deve estar em um formato específico. E o sistema deve saber tratar os erros de acordo.

Alguns erros tratados no sistema como, símbolo inexistente, símbolo duplicado, erro de sintaxe, erro de tipo de argumento de instruções, número de argumentos da instrução, e alguns outros serão mostrados no próximo capítulo.

Uma vez que o código foi montado com sucesso pode-se rodar através do botão *RUN*, como dito anteriormente. Caso o usuário gostaria de utilizar o resultado da montagem em outro simulador, ou então expor para uma *FPGA*, se pode clicar no botão *Assemble Results*, no menu global, do lado esquerdo da tela. e obter estes resultados. Exemplo pode ser visto na figura 3.2

Os resultados da montagem também podem ser visualizados em hexadecimal, ou formato MIF, como nas imagens 3.3 e 3.4

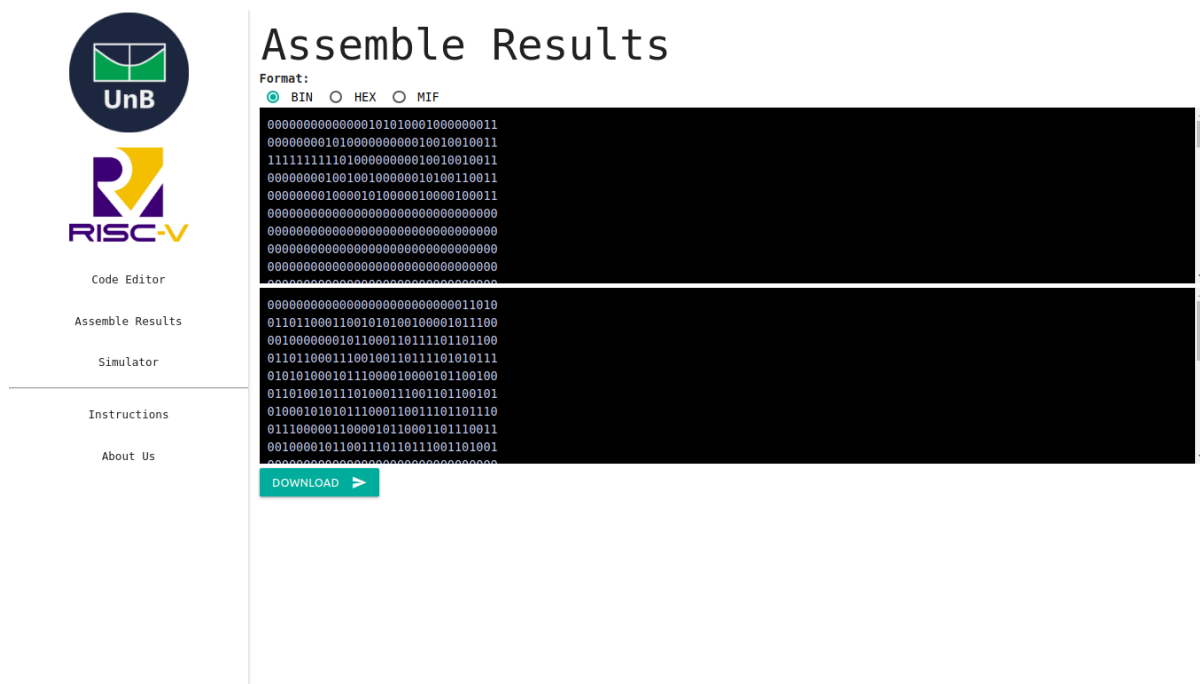


Figura 3.2: Resultados da montagem em binário.

3.5 Simulador

O simulador implementado neste projeto mostra o código montado, o código em hexadecimal, e as instruções que esses números em hexadecimal representam. Também pode se ver o mapa de memória e o estado dos registradores após o código montado ter sido executado ou em execução com a utilização do botão STEP. Lembrando que foi implementado a arquitetura RV32G, portanto dispõe-se de 32 registradores de 32 bits.

Na figura 3.5 é mostrado um exemplo dos resultados gerados pelo simulador implementado no projeto. O botão de RUN é utilizado para rodar o programa do momento em que está até o final da execução. Botão STEP irá executar apenas uma instrução. Pode-se utilizar o botão STEP para rodar algumas instruções e em seguida apertar RUN para executar até o final. A função de breakpoint não foi implementada. E o botão de RESET além de resetar as variáveis do sistema como os registradores, memória, program counter, também faz a montagem do código escrito na aba Code Editor.

A figura 3.6 é a continuação da tela de resultados. Esta mostra uma tela de output do sistema. Neste projeto foi implementado apenas duas funções SYSCALL. Uma é a impressão de um inteiro, e a outra o encerramento do programa.

Os valores de registradores e de mapa de memória podem ser visualizados na base decimal e hexadecimal.

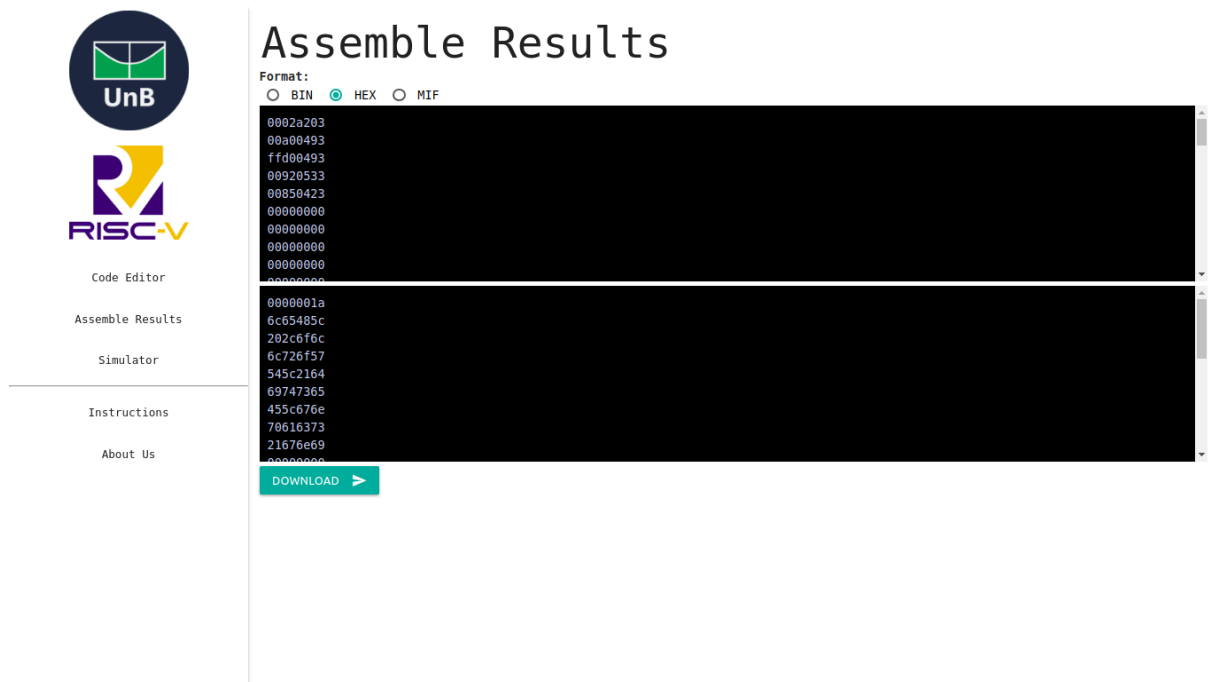


Figura 3.3: Resultados da montagem em hexadecimal.

3.6 Extensibilidade

Este projeto tem como objetivo poder ser extendido por outros interessados no estudo da arquitetura.

Por exemplo, para casos onde se deseja performance, podem ser conectados módulos através de extensões para python como cython [15], a instalação automática desses módulos não pode ser feita nessa versão.

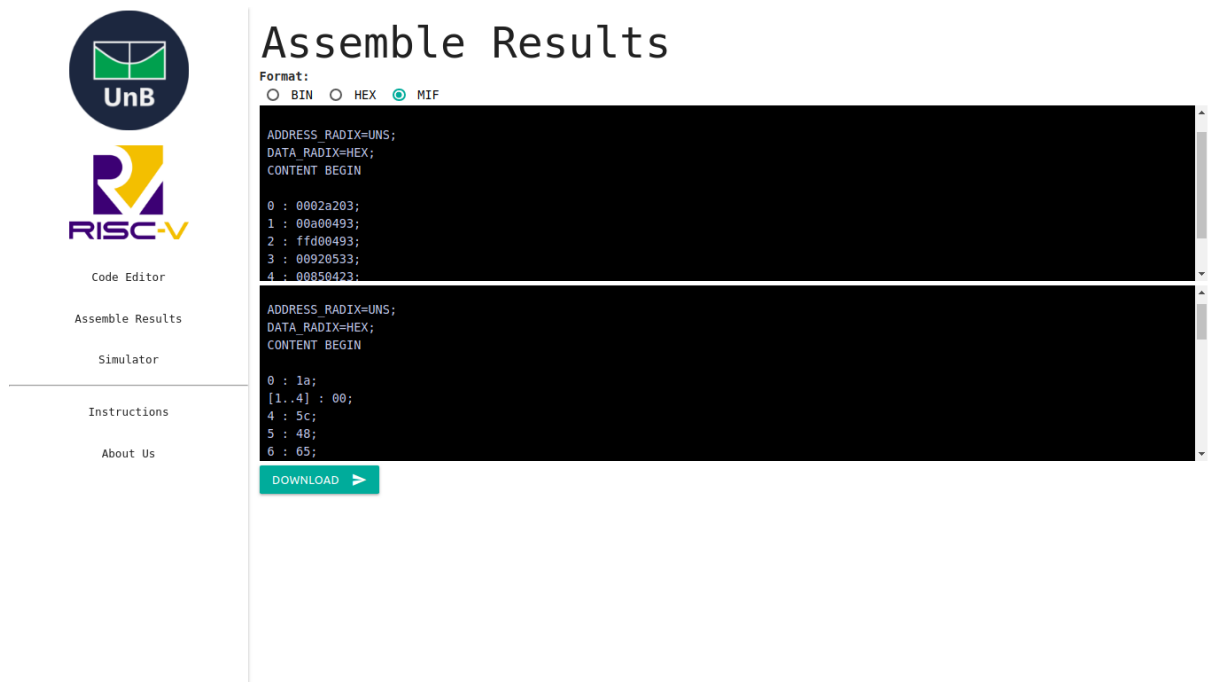


Figura 3.4: Resultados da montagem em formato MIF para FPGA.

UnB

RISC-V

Code Editor

Assemble Results

Simulator

Instructions

About Us

Simulator

RUN STEP RESET

assembled code

PC	Code	HEX	Instruction
0	0x0002a203	lw x4, x5, 0	
4	0x00a00493	addi x9, x0, 10	
8	0xffd00493	addi x9, x0, -3	
12	0x00920533	add x10, x9, x4	
16	0x00852423	sw x10, x8, 8	
20	0x00a00513	addi x10, x0, 10	
24	0x00000073	env x0, x0, 0	

memory map

☐ HEX ☒ DEC


	+0xc	+0x8	+0x4	+0x0
0x00000000	1819438935	23	1818576988	26
0x00000010	1885430643	1163683694	1769239397	1415324004
0x00000020	0	0	0	560426601

registers

☐ HEX ☒ DEC

Register	Value	Register	Value
x0	0	x16	0
x1	0	x17	0
x2	0	x18	0
x3	0	x19	0
x4	26	x20	0
x5	0	x21	0
x6	0	x22	0
x7	0	x23	0
x8	0	x24	0
x9	-3	x25	0
x10	10	x26	0
x11	0	x27	0
x12	0	x28	0
x13	0	x29	0
x14	0	x30	0
x15	0	x31	0

Figura 3.5: Resultados do simulador. Código montado, mapa da memória, registradores.




Code Editor
Assemble Results
Simulator
Instructions
About Us

```

16 0x00052423 sw x10, x8, 8
20 0x00a00513 addi x10, x0, 10
24 0x00000073 env x0, x0, 0

```

memory map

☒ HEX
☐ DEC

	+0xc	+0x8	+0x4	+0x0
0x00000000	0x6c726f57	0x00000017	0x6c65485c	0x0000001a
0x00000010	0x70616373	0x455c676e	0x69747365	0x545c2164
0x00000020	0x00000000	0x00000000	0x00000000	0x21676e69

console

```

>> Program terminated.

```

x2	0	x18	0
x3	0	x19	0
x4	26	x20	0
x5	0	x21	0
x6	0	x22	0
x7	0	x23	0
x8	0	x24	0
x9	-3	x25	0
x10	10	x26	0
x11	0	x27	0
x12	0	x28	0
x13	0	x29	0
x14	0	x30	0
x15	0	x31	0

Figura 3.6: Resultados do simulador. Console de saída de informações do sistema.

Capítulo 4

Resultados e Avaliação do Sistema

Neste capítulo mostraremos exemplos de resultados de utilização da ferramenta, para isso demonstraremos com alguns códigos simples a interação com o sistema e seus resultados, como também alguns exemplos de erros de montagem que são tratados neste projeto.

Para vermos melhores resultados, criamos os seguintes códigos,

- Aritmética simples
- Sequência de fibonacci

4.1 Possíveis erros de montagem

Como explicado na seção anterior, sendo a montagem, a parte que lida diretamente com a entrada do usuário, é necessário que o sistema saiba lidar com possíveis erros inseridos no código e devolver uma resposta adequada. Foram implementados as seguintes mensagens de erros,

Na figura 4.1, temos um exemplo de erro de declaração múltipla de símbolo, ou seja, se o programador declarar o mesmo símbolo em dois momentos diferentes no código.

Na figura 4.2, temos um exemplo de erro da diretiva `.ascii` ou `.string`, quando é declarado mais de uma string para um único símbolo.

Na figura 4.3, temos um exemplo de erro de valor de imediato, no caso deste projeto onde utilizamos tamanhos de palavra 32 bits, um valor superior a 4GB traria um overflow.

Na figura 4.4, temos um exemplo de erro de operando inválido, a instrução `ADDI` pede um imediato como último argumento, porém é fornecido um registrador.

Nas figuras 4.5, e 4.6, temos um exemplo de erro de operação não reconhecida, sendo que o primeiro mostra que foi fornecido uma instrução desconhecida, e no segundo uma diretiva desconhecida.

Code Editor

```
1 # RISC-V Assembly Code
2 .data
3     word teste: .word 26
4     str_teste: .ascii "\Hello, World!\nTesting\tEscaping!"
5
6 .text
7 lbl1:
8     addi x4, zero, 10
9
10 lbl1:
11     addi x4, zero, 10
12
```

ASSEMBLE ⚙️

Error: Duplicated Symbol. Line 11

RUN ⚙️

Figura 4.1: Erro de declaração múltipla de símbolo.

Code Editor

```
1 # RISC-V Assembly Code
2 .data
3     word teste: .word 1
4     str_teste: .ascii "\Hello, World!\nTesting\tEscaping!" "outra string"
5
6 .text
7     addi x4, zero, 10
8
```

ASSEMBLE ⚙️

Error: Incorrect number of string. Line:4

RUN ⚙️

Figura 4.2: Erro de declaração de diretiva string.

Na figura 4.7, temos um exemplo de erro de símbolo inexistente, o programador tentou utilizar um símbolo que não foi declarado antes.

Na figura 4.8, temos um exemplo de erro de tipo ou número de argumentos de diretiva, parecido com o erro da figura 4.2. Neste exemplo a diretiva `.word` pede um número e lhe é fornecido outra label.



Figura 4.3: Erro de imediato para o tamanho 32 de palavra 32 bits.



Figura 4.4: Erro de operando inválida, operando de tipo diferente do esperado.

4.2 Aritmética Simples

Este código mostra algumas instruções matemáticas básicas, de registrador para registrador, muito utilizadas em uma variedade de programas. Neste código poderemos ver a utilização destas instruções e seus resultados em cada registrador separado, verificando

Code Editor

```
1 # RISC-V Assembly Code
2 .data
3 word_teste: .word 26
4 str_teste: .asciiz "\Hello, World!\nTesting\tEscaping!"
5
6 .text
7 adds x4, zero, zero
```

ASSEMBLE ⚙️

Error: Operation adds not recognized. Line:7

RUN ⚙️

Figura 4.5: Erro de operação inválida, instrução não existente na arquitetura implementada.

Code Editor

```
1 # RISC-V Assembly Code
2 .data
3 word_teste: .wurd 26
4 str_teste: .asciiz "\Hello, World!\nTesting\tEscaping!"
5
6 .text
7 add x4, zero, zero
```

ASSEMBLE ⚙️

Error: Operation .wurd not recognized. Line:3

RUN ⚙️

Figura 4.6: Erro de operação inválida, diretiva não existente na arquitetura implementada.

suas funcionalidades.

Code Editor

```
1 # RISC-V Assembly Code
2 .data
3     word_teste: .word 26
4     str_teste: .asciiz "\Hello, World!\nTesting\tEscaping!"
5
6 .text
7     addi x4, zero, label
```

ASSEMBLE ⚙️

Erro. Símbolo inexistente. linha 7

RUN ⚙️

Figura 4.7: Erro de símbolo inválido, símbolo não foi declarado.

Code Editor

```
1 # RISC-V Assembly Code
2 .data
3     word_teste: .word abcd
4     str_teste: .asciiz "\Hello, World!\nTesting\tEscaping!"
5
6 .text
7     addi x4, zero, 10
8
```

ASSEMBLE ⚙️

Syntax Error: Wrong number or type of argument. Line 3

RUN ⚙️

Figura 4.8: Erro de argumentos de diretiva inválido, o tipo ou número de argumentos está incorreto.

4.2.1 Editor de código

Na figura 4.9 está o código que utilizamos para demonstração de algumas das instruções matemáticas mais básicas. Neste exemplo foi utilizada a instrução ADDI para inicializar valores nos registradores x3 e x4. A partir dos valores contidos nestes registradores realizamos várias operações matemáticas, ADD (adição), SUB (subtração), SRL (shift right logical), SLL (shift left logical), AND, OR, XOR, SLT (set less than).

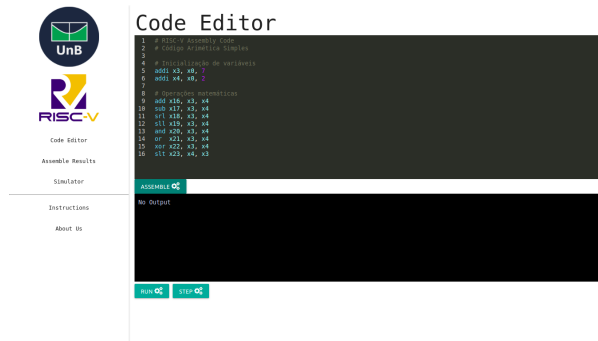


Figura 4.9: Demonstração de operações matemáticas básicas.

4.2.2 Simulação

Para cada operação realizada com estes valores, cada resultado foi armazenado em um registrador do x16 ao x23, como podemos ver na figura 4.10

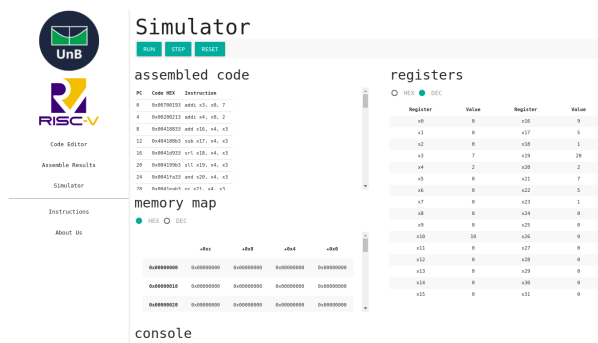


Figura 4.10: Demonstração de operações matemáticas básicas.

Como o programa não escreve na memória ou imprime resultados na tela, os resultados importantes a serem ressaltados são os valores de registradores e também o código montado. Os valores dos registradores são mostrados na forma decimal para visualização.

4.3 Sequência de Fibonacci

A sequência de fibonacci é um dos primeiros algoritmos que aprendemos quando iniciamos nos estudos de programação. Cada termo desta sucessão de números é gerada pela soma dos dois números antecessores.

4.3.1 Editor de código

Na figura 4.11 temos a inicialização de duas variáveis, `n_fibs`, que é a quantidade de termos da sequência que desejamos obter. E a outra variável é a `fib_base_mem_addr`. O valor atribuído a esta variável não é relevante neste programa, apenas denota a base de endereço onde serão armazenados os termos.

Depois temos a função `fib`, que inicializa registradores. Colocando em `x3` como o número de termos que serão escritos. Em `x4` o primeiro termo de fibonacci. E em `x6` um valor auxiliar. Em `x10` atribuímos o valor 1 para chamadas de impressão de inteiros através de `syscalls`

Ainda podemos ver uma parte da função `loop`, que irá calcular novos termos. A primeira instrução `BEQ` serve para verificarmos se já atingimos o número de termos desejados. Após a verificação faz-se um salto para funções de impressão. Vemos também a utilização do registrador `x7`, que servirá de auxiliar para não perdemos o valor de atual do termo de fibonacci. E então o registrador `x4` recebendo a soma do próprio `x4` que é o valor atual do termo de fibonacci e `x6` que é o valor auxiliar.



Figura 4.11: Primeira parte do código que gera sequência de fibonacci.

Na continuação da função loop, mostrado na figura 4.12, somamos o valor em x7, valor do termo anterior, com 0 e adicionamos em x6, variável auxiliar de fibonacci.

Após encontrar os termos terem sido atualizados para o próximo laço se faz um decremento do valor do registrador x3, que contém a quantidade de termos que o usuário deseja, que foi obtido a partir da label n_fibs.

Depois de decrementado o valor, se faz um salto para o início do loop, onde vai ser verificado se já foram encontrados todos os valores desejados e se pode pular para a label end e encerrar o programa.

Em seguida, nas linhas 25 e 30, estão as labels para as chamadas das rotinas de impressão chamadas dentro da função loop, print_to_console, e print_to_mem. Estas rotinas não utilizam a pilha para variáveis e retornos, como se utilizassem apenas variáveis globais.

A função print_to_console simplesmente move o valor de x4, termo de fibonacci, para o registrador x5, que é utilizado dentro da instrução ECALL (environment call). E então retornamos ao loop com a instrução JALR.

A outra função, print_to_mem, faz um SW (store word) do valor contido no registrador x4 no endereço dado pela soma do valor de x30 adicionado do endereço da variável de fib_base_mem_addr. Depois adicionamos 4 em x30 para que no próximo laço seja impresso o próximo inteiro no endereço de store word correto. E então retornamos ao loop.



The screenshot shows the RISC-V Code Editor interface. On the left, there is a sidebar with the 'UnB' logo and the 'RISC-V' logo. Below the logos are links for 'Code Editor', 'Assemble Results', 'Simulator', 'Instructions', and 'About Us'. The main area is titled 'Code Editor' and contains the following assembly code:

```

19 add x4, x4, x6 # fib = fib+fib aux
20 addi x6,x7,0 # fib_aux = aux
21
22 addi x3,x3, -1 #dec counter
23 jal x0, loop
24
25 print_to_console:
26 addi x5,x4,0
27 ecall x0,x0,0 # print value
28 jalr x0, x20, 0
29
30 print_to_mem:
31 sw x4, x30, fib_base_mem_addr
32 addi x30, x30, 4
33 jalr x0, x21, 0
34
35 end:
36 |
37

```

Below the code editor, there is a green 'ASSEMBLE' button with a gear icon. Below that is a black console area with the text 'No Output'. At the bottom of the console area, there are two buttons: 'RUN' and 'STEP', both with gear icons.

Figura 4.12: Segunda parte do código que gera sequência de fibonacci.

Ao final do código devemos ter o número de termos de fibonacci fornecidos pelo usuário mostrado tanto na console de saída quanto em endereços na memória.

4.3.2 Simulação

Na primeira parte dos resultados demonstrados na figura 4.13 vemos parcialmente o código montado gerado pelo montador, valores residuais dos registradores utilizados e os valores da memória.

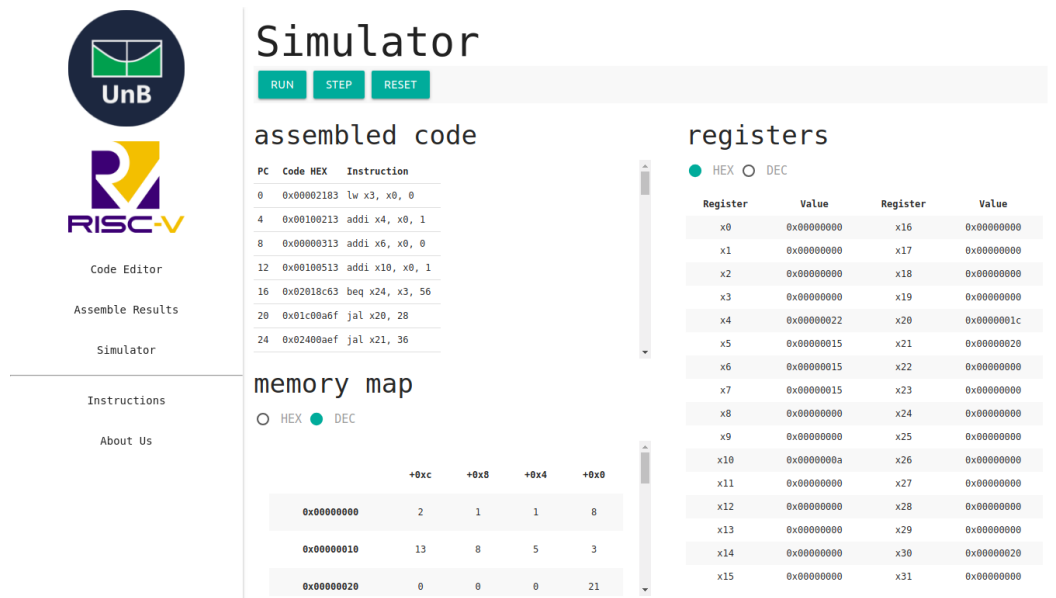


Figura 4.13: Primeira parte dos resultados gerados da sequência de fibonacci.

Na primeira instrução vemos que o programa executa um Load Word no endereço 0 da memória, este endereço contém o valor fornecido pelo programador com o número de termos de fibonacci que gostaria de obter. A partir do segundo endereço, o 0x00000004, temos a sequência de valores de fibonacci impressos na memória a cada iteração do loop.

Na figura 4.14 podemos ver a console de saída. Esta parte nos mostra a funcionalidade da chamada de ambiente, ou SYSCALL, que imprime um número inteiro na tela.

Os termos foram impressos um em cada linha, cada linha representa uma chamada da instrução ECALL com o valor 1 no registrador x10, e o termo a ser impresso contido no registrador x5. E ao final do código temos a outra chamada de ambiente implementada que é a de saída do programa, chamando ECALL com o valor 10 no registrador x10.



Figura 4.14: Segunda parte dos resultados gerados da sequência de fibonacci.

Capítulo 5

Conclusões

5.1 Objetivos atingidos

Neste projeto conseguimos realizar um primeiro estudo da arquitetura RISC-V, e implementar uma interface de fácil utilização, multiplataforma, onde um usuário pode escrever códigos na linguagem assembly RISC-V, realizar a montagem do código, e rodar uma simulação deste código.

O sistema permite exportar as saídas do montador para que o usuário possa simular seu código em outras ferramentas de simulação, ou uma implementação própria em uma *FPGA*.

Os módulos de montagem e simulação foram implementados como bibliotecas separadas, por isso podem ser utilizadas em outros contextos. Por exemplo, em linha de comando.

5.2 Pontos positivos e negativos

Alguns pontos positivos da ferramenta são aqueles já citados quando descrevemos as vantagens de se utilizar uma plataforma web, e alguns outros,

- Não necessidade de instalação e configurações para começar a utilizar a ferramenta
- Multiplataforma
- Extensibilidade

Desvantagens dessa solução hospedada são,

- Custo de hospedagem
- Possível indisponibilidade

Porém, apesar dessas desvantagens citadas acima, o fato da solução ser aberta diminui o peso dessas desvantagens, pois o código pode ser baixado e rodado localmente.

Assim, mesmo que haja algum problema de hospedagem, pode-se obter facilmente o código fonte, e tendo um interpretador instalado em seu computador, se pode fazer sua própria hospedagem local, pois o interpretador já fornece um servidor web imbutido nativamente para desenvolvimento.

5.3 Dificuldades

No começo do projeto tentou-se implementar um editor de texto em javascript sem utilização de bibliotecas terceiras, porém a tarefa se mostrou muito difícil de ser realizada, então optou-se por utilizar a biblioteca CodeMirror, muito utilizada por vários sites populares de desenvolvimento.

Na parte do montador foi onde houve o maior esforço. A parte documentada que diz respeito a linguagem de programação ainda é escassa, porém para a solução limitada que foi feita neste projeto foi o suficiente. Por ser o módulo que interage diretamente com a entrada do usuário, este se mostrou ser o ponto crítico da solução.

O simulador foi bem simplificado, por isso não houveram grandes dificuldades. O desenvolvimento do simulador tinham algumas dificuldades de saber como as instruções por exemplo de "branch" funcionavam à nível de hardware, pelo tipo de codificação utilizada, pois algumas informações sobre estas não ficam explícitas nas documentações.

Grande parte do esforço também foi direcionado à integração do frontend com o backend, pois pode-se considerar dois sistemas diferentes. O que era um dos objetivos, para que a ferramenta seja flexível, e possa ser estendida e evoluída mais a frente. Por isso foi decidido utilizar uma API para que seja flexível utilizar novas interfaces ou novos montadores, simuladores customizados.

5.4 Melhorias e trabalhos futuros

Existem muitas funcionalidades a serem feitas, citando algumas,

- Expandir arquitetura para 64-bits
- Sistema de login/logout, com um sistema de usuários, poderíamos montar um sistema para salvar e compartilhar códigos com outros usuários. Também inserir módulos personalizados.
- Contador de instruções.

- Bitmap mapping, para termos uma representação visual da memória do sistema.
- Adição de instruções e pseudo-instruções customizadas.
- Utilização de bibliotecas ou frameworks mais modernos para frontend como ReactJS, AngularJS, VueJS.
- Melhorar interface para dispositivos móveis.

A parte de extensões da solução não foi realizada, porém teria muitas aplicações e diferenciaria bastante de outras ferramentas. Para isso precisaríamos ter uma documentação bem especificada, para que haja compatibilidade da interface entre os módulos de extensão.

Uma das funcionalidades que seria muito interessante ter realizado no projeto seria a possível extensão de códigos em C, para podermos utilizar modelos descritos em mais baixo nível feitos por exemplo com a biblioteca *SystemC*. Com isso poderíamos testar com maior facilidade implementações baseados em performance.

Referências

- [1] Waterman, Editors Andrew e RISC V Foundation Krste Asanovic: *The risc-v instruction set manual, volume i: User-level isa, document version 2.2*. <https://riscv.org/specifications/>, May 2017. 1, 6, 8, 10, 11
- [2] Waterman, Andrew: *Design of the RISC-V Instruction Set Architecture*. Tese de Doutoramento, EECS Department, University of California, Berkeley, Jan 2016. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>. 2, 8
- [3] Armasu, Lucian: *Big tech players start to adopt the risc-v chip architecture*. <https://www.tomshardware.com/news/big-tech-players-risc-v-architecture,36011.html>, November 2017. 4
- [4] NVIDIA: *Nvidia risc-v story*. https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf, July 2016. 4
- [5] Andrew WatermanYunsup LeeDavid PattersonKrste Asanovic: *The risc-v instruction set manual, volume i: Base user-level isa*. <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf>, May 2011. 8
- [6] Andrew WatermanYunsup LeeDavid PattersonKrste Asanovic: *The risc-v instruction set manual, volume i: Base user-level isa*. <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf>, May 2014. 8
- [7] Foundation, RISC V: *Sobre a risc-v foundation*. <https://riscv.org/risc-v-foundation/>, 2018. Accessed: 2018-06-20. 8
- [8] Foundation, RISC V: *Membros do conselho risc-v foundation*. <https://riscv.org/leadership/>, 2018. Accessed: 2018-06-20. 9
- [9] Foundation, RISC V: *Frequentlyl asked questions*. <https://riscv.org/faq/>, 2018. 9
- [10] Technology, Graz University of: *Meltdown and spectre exploits*. <https://meltdownattack.com/>, 2018. Accessed: 2018-06-30. 9
- [11] *Two security flaws in modern chips cause big headaches for the tech business*. <https://www.economist.com/science-and-technology/2018/01/04/two-security-flaws-in-modern-chips-cause-big-headaches-for-the-tech-business>. Accessed: 2018-06-20. 9

- [12] Krste Asanović, Rick O'Connor: *Building a more secure world with the risc-v isa*. <https://riscv.org/2018/01/more-secure-world-risc-v-isa/>, 2018. Accessed: 2018-06-20. 10
- [13] Mikowski, Michael S e Josh C Powell: *Single page web applications*. B and W, página 43, 2013. 17
- [14] Haverbeke, Marijn: *Codemirror*. <http://codemirror.net/>, July 2018. 17
- [15] *Cython c-extensions for python*. <http://cython.org/>. Accessed: 2018-06-20. 20