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
Stanford University

Dr. Leslie Lamport
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 utililzar a classe LaTeX 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`). 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 LaTeX 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

Lista de Figuras

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 computadoers 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 performace:

- A ISA é exageradamente otimizada para o padrão de pipeline de cinco estágios em ordem, como jumps e branches são atrasados, isso complica implementações superescalares 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. Históricamente, 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 técnicamente 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 Henessy.

1.2 Importância acadêmica e industrial do RISC-V

RISC-V foi desenvolvido por Krste Asanoviü, 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 industria 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 rasperry 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 utizado para propósitos gerais.
- Os módulos não-padrões são módulos especializados, podendo conflitar com outros módulos. A previsão é de que no futuro hajam 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 disvantagens 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 ManualVolume 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 STMicroeletronics 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,

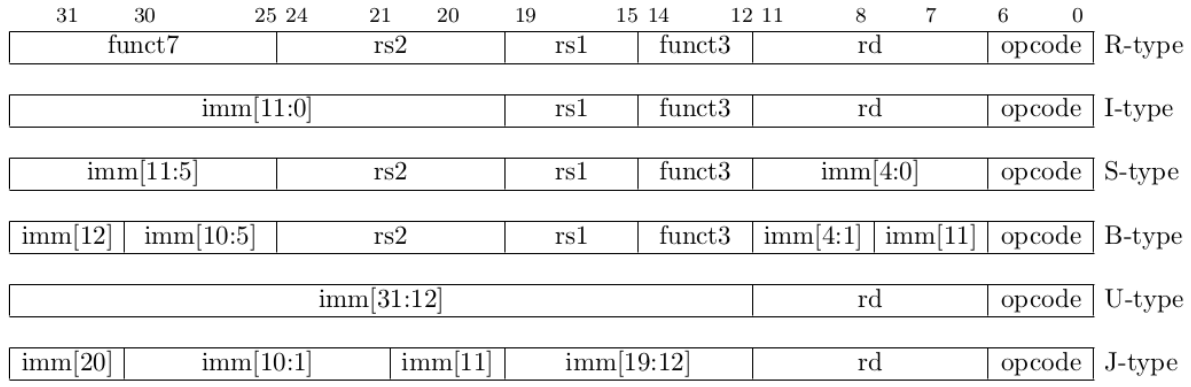

Figura 2.1: Formatos de instruções da ISA básica
Fonte: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2 [1]

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 extendidos pelo sinal, exceto pelos imediatos das instruções de CSR(*"Control and Status Registers"*).

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

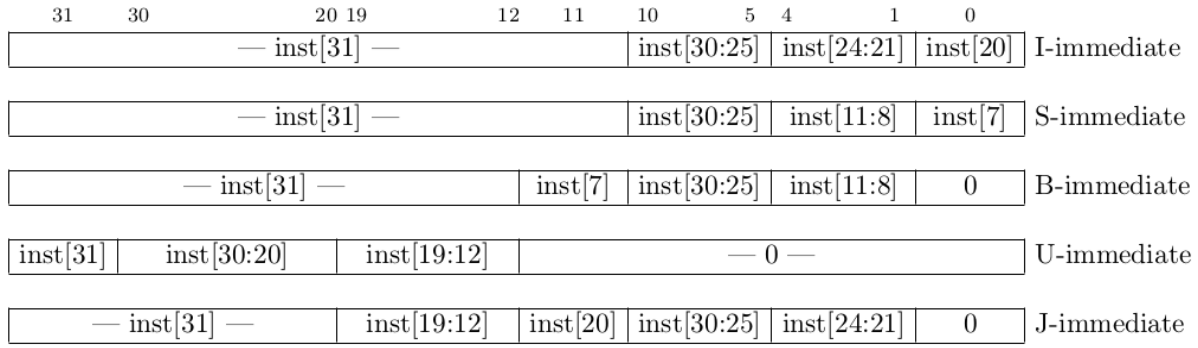

Figura 2.2: Encoding dos imediatos de cada tipo de instrução
Fonte: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2 [1]

As instruções do tipo R, são as do tipo registrador-registrador, utilizam três 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 instruções incluem, AND, OR, SLTU, SRL, SRA...

As do tipo I, são as do tipo registrador-imediato, são 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 da 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 da pelo conteúdo de x1 mais o imm
- SW x1, x2, imm. Store Word. Armazena a word contida em x2 no endereço que se da 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 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.

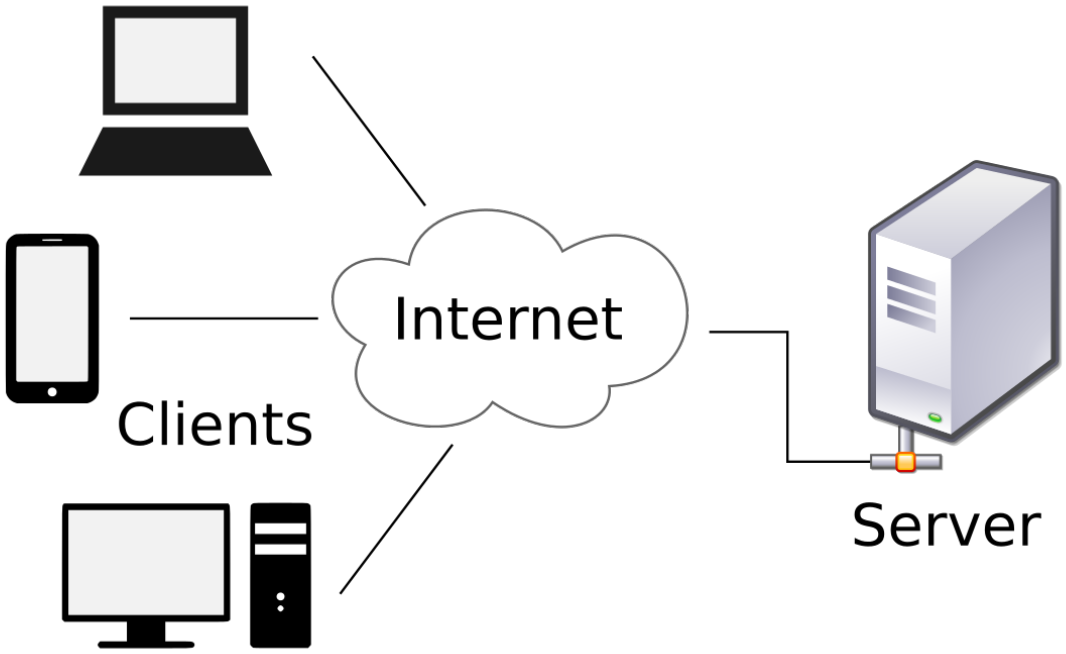

Figura 2.3: Ilustração da arquitetura Cliente-Servidor
Fonte: https://upload.wikimedia.org/wikipedia/commons/c/c9/Client-server-model.svg

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 dinâmicamente 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 decido 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.

Abaixo iremos descrever por módulos cada componente do sistema.

3.2.1 Front-end

O front-end da aplicação como já explanado anteriormente é a parte visual da aplicação. É a parte do software que lidará diretamente com a usabilidade e experiência do usuário. Nesta seção será abordado como foi implementado com maiores detalhes dentro da solução apresentada neste projeto.

Single Page Application

Uma única página com todos os componentes carregados, porém escondidos. Com isso evitamos recarregar a página várias vezes, pois a quantidade de dados que teríamos que enviar e receber do servidor web seria alta, além de termos que processar esses dados para manter o estado atual do sistema sempre sincronizado.

Na parte de front-end temos três módulos ou seções. Estes módulos são definidos por uma estrutura em HTML utilizando a tag section com ID's. Desse jeito podemos separar visualmente com CSS e funcionalmente com javascript. As seções existentes são entrada, saída e simulator.

Cada seção pode ser acessada pelo menu global da aplicação como mostrado na figura 3.1.

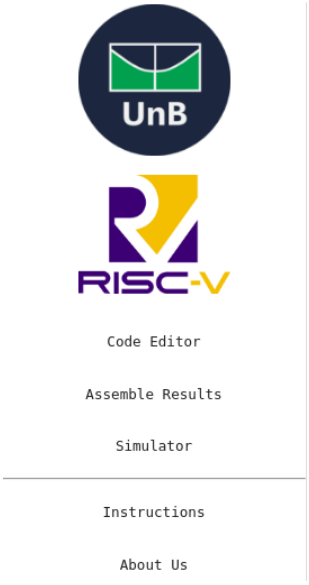

Figura 3.1: Menu global do sistema. Principal componente de navegação.

Neste menu vemos duas partes separadas por uma linha horizontal, a parte superior contém as seções do sistema, enquanto a parte inferior contém seções de informações.

Para a lógica do front-end temos dois módulos em jQuery, o riscv_flow e riscv_functions. Detalharemos estes módulos abaixo.

riscv_flow Este módulo trata de associar eventos aos botões do sistema para esconder e mostrar seções através de alterações no CSS. E também associa os botões de ações do sistema.

Por exemplo, para mostrarmos a seção de simulação, escondemos a seção de entrada e a seção de saída e mostramos a seção de simulação, e então para mostrar outra seção mostramos essa seção e escondemos a de simulação.

A associação dos eventos de comunicação com o back-end também é feito no riscv_flow.

Por exemplo, ao clicarmos no botão de montagem, a função que associa o clique no botão com a funcionalidade de montagem é escrita aqui. Porém a funcionalidade da comunicação em é feita no módulo riscv_functions, que iremos detalhar no próximo parágrafo.

riscv_functions São implementadas as funcionalidades e eventos de comunicação com o back-end. É responsável por atualizar informações na página como mensagens de resposta da montagem, no caso do montador, ou valores de memória e registradores no caso do simulador. Esta atualização ocorre sem fazer reload da página.

Neste módulo estão funções de conversão de base, e tratamento de dados. Essas funções são importantes para facilitar a visualização dos dados pelo usuário.

As duas funcionalidades principais deste módulo são as funções de assemble e run_simulation.

Estas duas funções são descritas aqui e associadas aos botões no módulo riscv_flow. Ambas utilizam o mecanismo conhecido como *AJAX* do inglês *"Asynchronous Javascript And XML"*. Este mecanismo utiliza um objeto nativo de browsers modernos que serve é utilizado para fazer requisições de dados de Web Servers. E é ele que permite trocar informações com o servidor sem precisar recarregar a página inteira.

No caso das funções assemble e run_simulation, eles fazem requisições POSTs para o nosso back-end.

A função assemble manda o código escrito na seção de edição de código e recebe as informações de montagem, memória de código, memória de dados, e erros. Também insere as informações de programa na seção de simulação.

A função run_simulation envia todas as informações do contexto da execução no momento, a memória de código, e de dados, valores de registradores, contador de programa, entradas e saídas de usuário, e também um inteiro que diz quanta instruções o simulador deve executar. Este recebe praticamente as mesmas informações porém atualizadas uma instrução depois da execução.

3.2.2 Back-end

Nesta seção iremos abordar as implementações das funcionalidades do sistema, cada componenete do back-end e como se comunicam.

main

O primeiro componente a ser detalhado é o componente main. Este é o componente que liga o front-end com o back-end, pois recebe as chamadas do front-end e executa os componentes requisitados.

Por exemplo, ao clicar no botão ASSEMBLE o usuário está enviando uma requisição POST com o código ASSEMBLY RISC-V ao componente MAIN, e este irá chamar o componente ASSEMBLER. Após a execução do componente ASSEMBLER, este irá devolver ao front-end as informações devidas.

O componente MAIN responde à três requisições HTTP:

- Requisição GET "/". Esta requisição irá devolver a página completa do front-end ao usuário. É a página inicial do sistema. A requisição é feita ao entrar com a URL no navegador.
- Requisição POST "/assemble". Esta requisição é feita quando o usuário clica no botão ASSEMBLE na tela do editor de código, ou no botão RESET na tela de

simulação. E retorna o código montado, memória, erros, ou outras informações de montagem.

- Requisição POST "/run". Esta requisição pode ser feita ao clicar no botão RUN ou STEP, estes botões existem tanto na tela do editor de código quanto na tela de simulação. Ao realizar o clique em algum dos botões, o usuário estará enviando todas as informações atuais da simulação, isso inclui, o código montado, valores de memória, registradores, contador de programa, valores de entrada e saída, e também um diferenciador do botão step ou run, para o programa saber se irá rodar até o final ou apenas um passo.

Após chamar os componentes adequados à cada requisição feita pelo usuário, o componente MAIN irá retornar as informações em formato JSON para a função javascript que fez a requisição, e então mostrar esses dados no front-end. As informações que cada requisição retorna serão melhor detalhadas nas próximas seções.

utils

O componente UTILS agrega funções e variáveis auxiliares aos outros componentes, por exemplo, funções de conversão de base, codificação e decodificação de instruções, execução de instruções, e configurações, como tamanho da das memórias de código e dados.

settings Neste módulo estão contidos definições como nomes de registradores, suas representações em binário, tamanho das memória de dados e código em bytes, tamanho da palavra, que no caso deste sistema é 32 bits, variáveis de decodificação, opcode, rd, rs1, rs2, imediatos entre outros. Também está definido neste módulo um número máximo de instruções rodados por simulação, para evitar laços infinitos e sobrecarga.

utilities No módulo UTILITIES estão definidas várias funções que dão suporte ao funcionamento do sistema, como funções de conversão de base, de verificação de tipo, e algumas funções para debug, para mostrar registradores, região da memória entre outros.

instructions Este módulo contém a definição de todas as instruções implementadas do sistema, contém várias tabelas para codificação e decodificação que são utilizadas para a montagem e para a simulação do sistema.

As tabelas principais são as tabelas instruction_table e reverse_instruction_table. A instruction_table é feita no formato,

```
# Instruction Table
opcode:
```

```
{
    "type": tipo,

    funct3:
    {
         funct7 : nome_da_instrucao
    }
}

# Exemplo

# JALR
"1100111" :
{
    "type": "i",
    "000" : "jalr"
},

# ADDI, SLTI, SLTIU XORI ORI ANDI SLLI SRLI SRAI
"0010011" :
{
    "type": "i",
    "000": "addi",
    "010": "slti",
    "011": "sltiu",
    "100": "xori",
    "110": "ori",
    "111": "andi",
    "001": "slli",
    "101": "sri"
},
```

E a reverse_instruction_table é feita no formato,

```
# Reverse Instruction Table
nome_da_instrucao:
{
    "type": tipo,
```

```
    "size": tamanho_em_bytes,
    "opcode": opcode,
    "funct3": valor_funct3,
    "funct7" : valor_funct7
}

# Exemplo

    "lui" : {
        "type":"u",
        "size":4,
        "opcode":"0110111"
        },
    "auipc" : {
        "type":"u",
        "size":4,
        "opcode":"0010111"
        },
    "jal" : {
        "type":"uj",
        "size":4,
        "opcode":"1101111"
        },
```

Também neste módulo estão escritas as implementações das instruções, constituídas de funções que simulam seus comportamentos e uma tabela de execução de instrução, no formato abaixo

```
# Instruction Execution Table
instruction_execution_table = {
    nome_da_instrucao1: nome_da_funcao1,
    nome_da_instrucao2: nome_da_funcao2,
    nome_da_instrucao3: nome_da_funcao3,
    ....
    nome_da_instrucaox: nome_da_funcaox
}

# Exemplo
```

```
instruction_execution_table = {
    "lui" : instr_lui,
    "auipc" : instr_auipc,
    "jal" : instr_jal,
    ...
    "csrrwi" : instr_csrrwi,
    "csrrsi" : instr_csrrsi,
    "csrrci" : instr_csrrci,
    "nop": instr_nop
}
```

assembler

Detalharemos agora o componente do montador, o ASSEMBLER. Este componente recebe do componente MAIN o código fonte do programa escrito na tela de editor de código. A partir deste código fonte, é gerado o código máquina da aplicação, ou então retornado os erros que foram inseridos no código que impossibilitou a montagem do programa. Caso o montador tenha sido executado com sucesso, o componente irá retornar o código montado em binário, e os dados de memória declarados no código fonte.

Como mencionado anteriormente no capítulo 2, neste projeto utilizamos o algoritmo de duas passagens. Explicaremos cada parte da implementação do montador mais abaixo.

Abaixo descreveremos as funções implementadas para realizar esta tarefa.

assemble A função assemble inicializa as variáveis que serão utilizadas, chama as funções first_pass, se tudo ocorreu bem, chama a função second_pass.

Após o término da montagem a função irá verificar se houve erros, caso tenha, a função irá gerar lista de erros e warnings.

Antes de retornar as informações para a função chamante, esta função irá resetar as variáveis globais, formatar as informações para retorno e só então retorna.

first pass

second pass

check operands

split tokens

simulator

O componente SIMULATOR é o que lida com as requisições /run, tanto para rodar o programa completo ou apenas um passo de cada vez. É o componente principal para a simulação do código montado pelo ASSEMBLER.

run Este módulo recebe vários parâmetros, que basicamente consistem o estado atual da execução, na maioria dos casos são estados iniciais rodando até o final, porém para os runs que executam apenas uma instrução é necessário saber o estado dos registradores e memória na instrução anterior.

Os parâmetros que ele recebe são,

- code. É o código máquina gerado pelo ASSEMBLER.
- memory. Memória de dados.
- registers. Todos os valores dos registradores.
- pc. Contador de programa.
- console_input. Se houvesse um syscall com entrada de dados pelo usuário utilizando o teclado, este argumento seria passado através desta variável.
- console_output. Para a saída de dados, no nosso sistema podemos ver o resultado da impressão de inteiros por exemplo, ou então ao encerrar o programa. É enviado para manter uma consistência com as próximas mensagens.
- step_count. Número de instruções que se deseja executar. Utilizando a interface web não se pode executar mais de uma instrução por requisição, apenas se executá-lo por completo.

A primeira coisa que a função RUN faz é inicializar e formatar as variáveis recebidas, incluindo em listas na maioria das vezes. Com todas as variáveis setadas e o estado atual da execução está atualizada o programa entra em um laço e só sairá caso uma das quatro condições se satisfazer,

- Número máximo de ciclos. Caso se atinja essa condição, significa que o programa é muito grande ou talvez esteja em laço infinito.
- Contador de steps zerado. Caso seja apenas uma instrução, o programa irá sair com essa condição para que não execute até o final.

- SYCALL de encerramento de programa. Caso tenha a instrução seja um SYSCALL com os argumentos de saída de programa
- Erro. Algum erro de execução ocorreu.

Dentro do laço irão ser executadas três funções, que serão detalhadas mais abaixo. A função fetch, para buscar na memória de código a instrução. A função decode, para saber qual a instrução e qual o contexto dela. E finalmente a execute que irá rodar a função adequada para a instrução dada.

Ao final, o resultado da execução é formatada e as variáveis globais resetadas.

fetch Esta função verifica se o contador de programa pode ser índice de uma posição da memória de código, caso possa, busca a instrução e incrementa o contador de programa e retorna a instrução. Caso o contador não possa ser índice, retorna -1 para que seja interpretado como erro.

decode Recebida a instrução, separa-se as informações de opcode, registradores, funções, e imediatos. Estes valores ficam em variáveis globais que serão utilizadas posteriormente na função de execução.

Após feito isso, a função busca na tabela de instruções qual o nome da instrução sendo executada e retorna este valor.

execute Tendo o nome da instrução, a função busca a entrada contida na tabela de execução de instruções e executa. Os valores necessários para a execução já estão todas contidas em variáveis globais decodificadas na função anterior.

Após a execução da instrução, se faz a atribuição do valor do registrador x0 para 0, pois como o valor de x0 é hard-wired, caso alguma instrução tenha alterado seu valor, neste momento o valor é alterado de volta para 0.

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 funcionaliades 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.

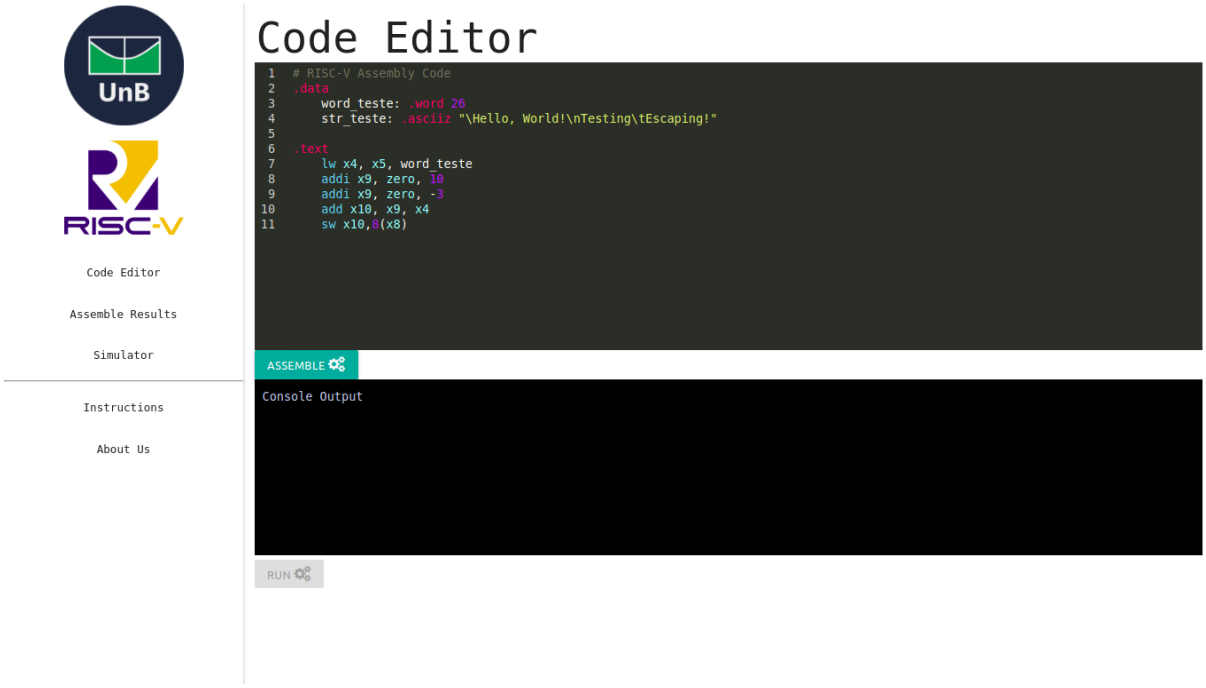

Figura 3.2: Página inicial, mostra o editor de texto com um código exemplo.

Na figura 3.2, 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 trocaad para a tela de simulação, que mostraremos nas próximas seções.

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 exporar 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 3.3

Figura 3.3: Resultados da montagem em binário.

Os resultados da montagem também podem ser visualizados em hexadecimal, ou formato MIF, como nas imagens 3.4 e 3.5

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 3.6 é mostrado um exemplo dos resultados gerados pelo simulador implementado no projeto. O botão de RUN é utilizado para rodar o programa do momento

Figura 3.4: Resultados da montagem em hexadecimal.

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 3.7 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.

3.6 Extensiblidade

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 modulos através de extensões para python como cython [15], a instalação automática desses módulos não pode ser feita nessa versão.

Figura 3.5: Resultados da montagem em formato MIF para FPGA.

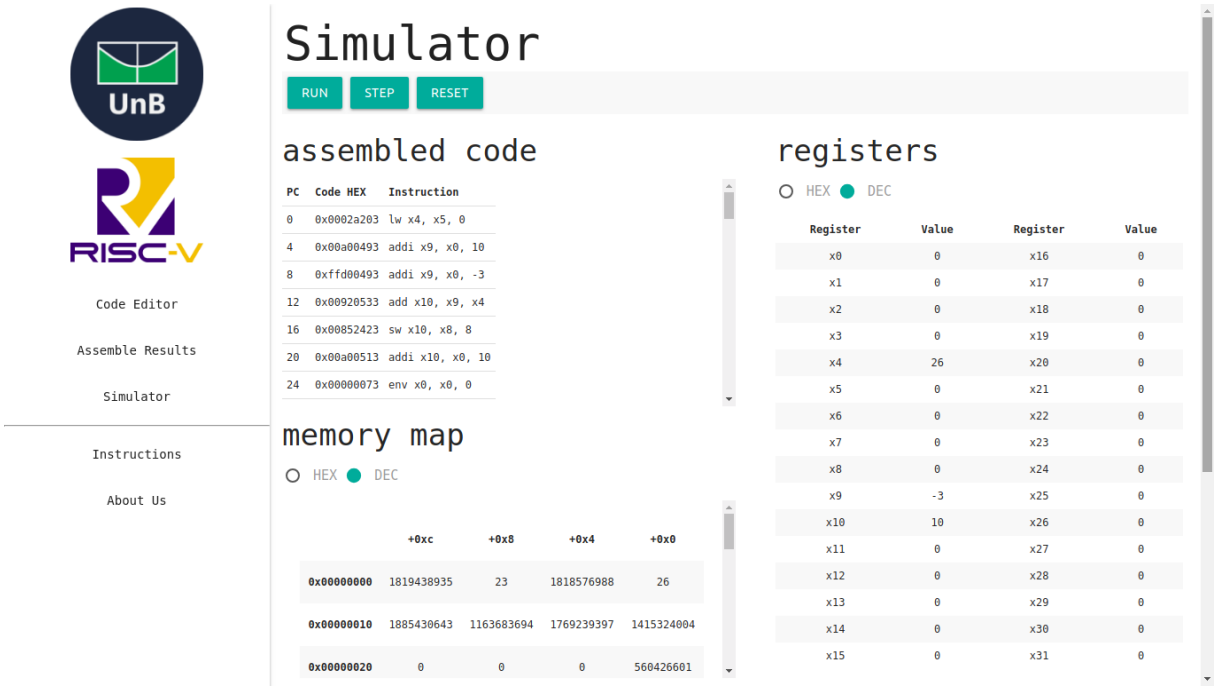

Figura 3.6: Resultados do simulador. Código montado, mapa da memória, registradores.

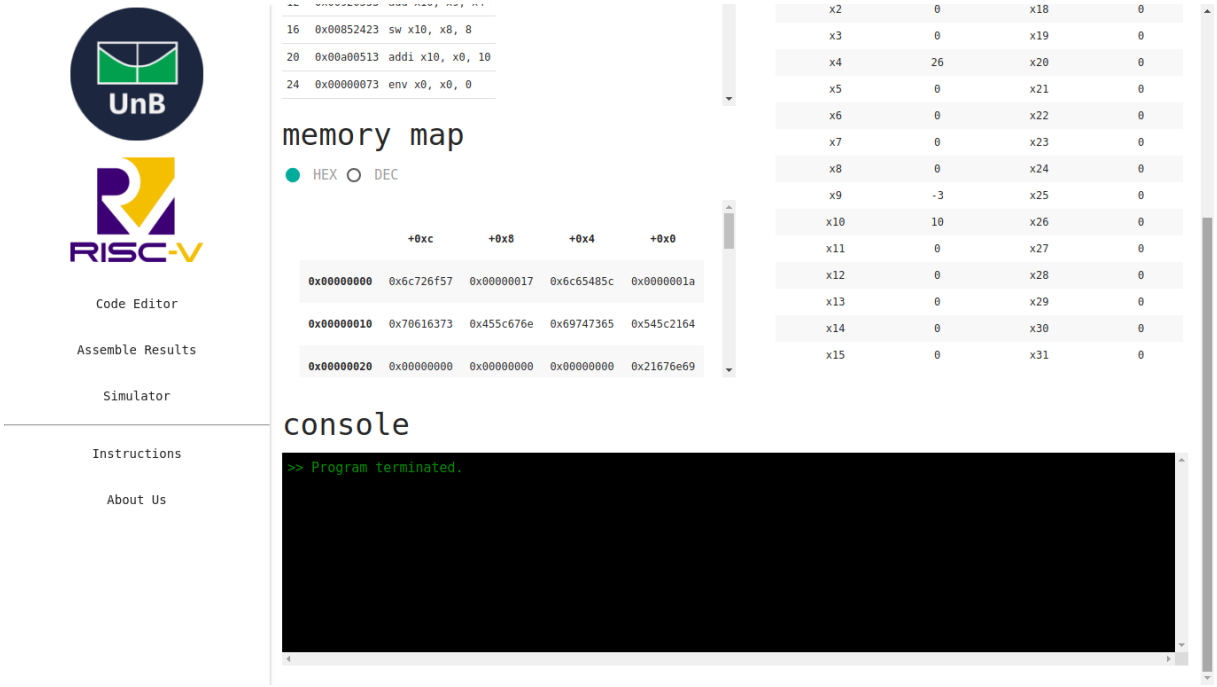

Figura 3.7: Resultados do simulador. Console de saída de informações do sistema.

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 menasgens 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 .asciiz 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.

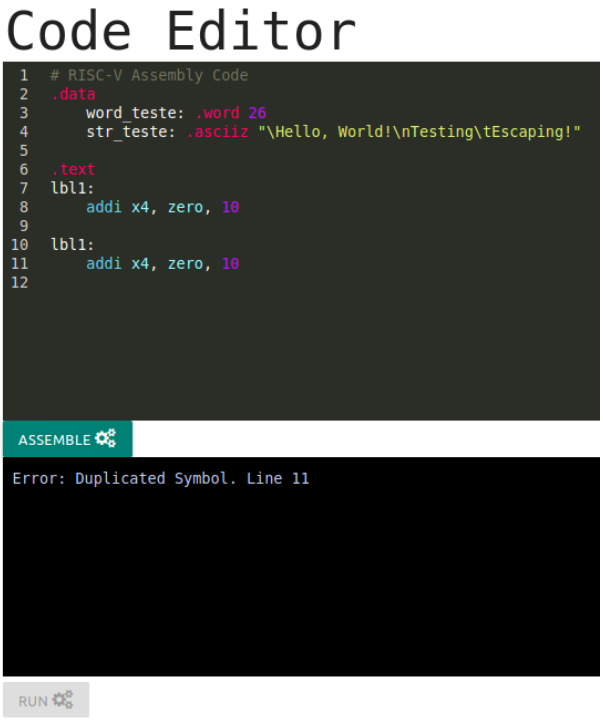

Figura 4.1: Erro de declaração múltipla de símbolo.

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

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

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

suas funcionalidades.

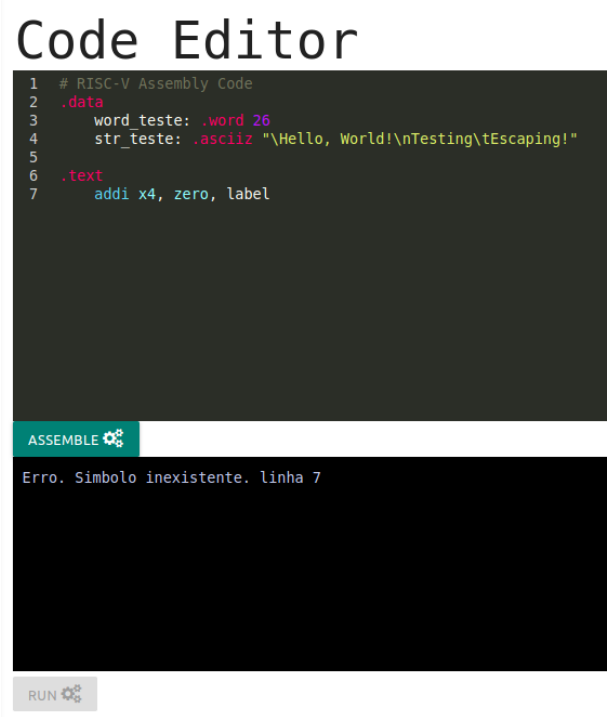

Figura 4.7: Erro de símbolo inválido, símbolo não foi declarado.

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 utilziada 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).

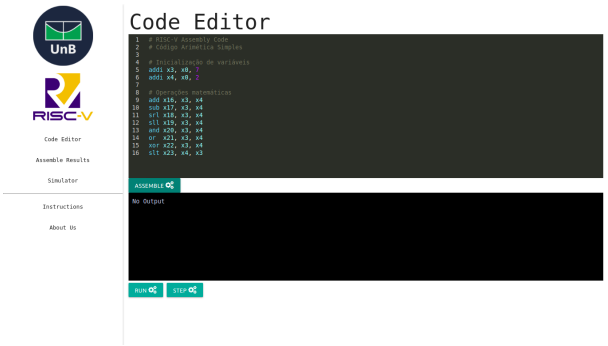

Figura 4.9: Demonstração de operações matemáticas básicas.

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

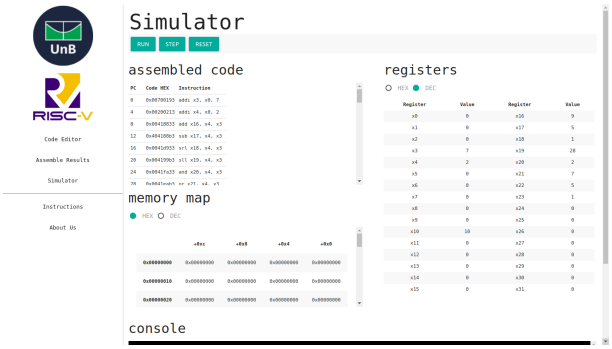

Figura 4.10: Demonstração de operações matemáticas básicas.

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 antescedentes.

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 atribuido 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 atribuimos 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 gloabais.

A função print_to_console simplesmente mvoe 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.

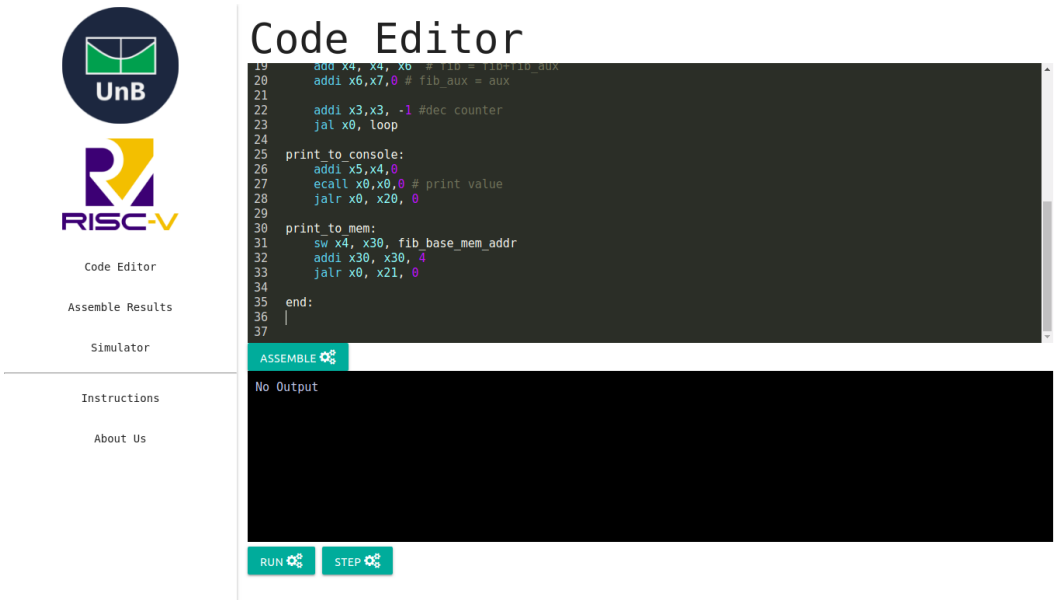

Figura 4.12: Segunda parte do código que gera sequência de fibonacci.

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 parcialemnte o código montado gerado pelo montador, valores residuais dos registradores utilizados e os valores da memória.

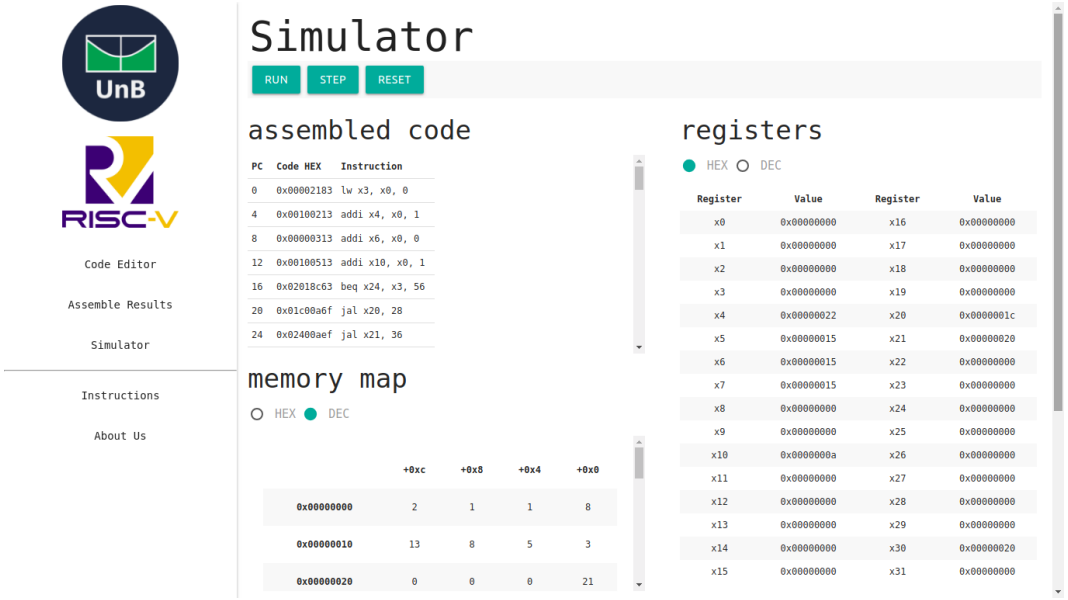

Figura 4.13: Primeira parte dos resultados gerados da sequência de fibonacci.

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 gue 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 biblitoecas 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 utilzar 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 explicitas 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 extendida e evoluida 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, poderiamos 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 performace.

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: *Frequentyl 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. 26

[15] *Cython c-extensions for python.* `http://cython.org/`. Accessed: 2018-06-20. 28