Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

O Processo de Desenvolvimento de uma CPU RISC-V

Elisa Uhura Pereira da Silva

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman

São Paulo
2021

Check the circuit!
– Spock

Agradecimentos

Agradeço o apoio e motivação de todos que me incentivaram a completar este trabalho, em especial Carime, Fujiwara, Goldman e Luana. Também, pelo do apoio do CarlosEDP e da comunidade RISC-V Brasil, que foram uma fonte de conhecimento inestimável. Por fim, agradeço minha família, que me acompanhou nas longas jornadas de trabalho.

Resumo

Elisa Uhura Pereira da Silva. **O Processo de Desenvolvimento de uma CPU RISC-V**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

A RISC-V é uma arquitetura de conjunto de instruções aberta e disponível de forma gratuita para uso na indústria e na academia. Com o objetivo de ser a arquitetura padrão para todos os dispositivos computacionais, sua diferença principal é a disponibilidade de diversas implementações abertas desenhadas para uso em diversos tipos de tarefas computacionais. Este trabalho apresenta o processo de desenvolvimento de um processador usando a arquitetura RISC-V com enfoque na parte de verificação do circuito. O processo foi divido nas etapas de *planejamento e preparação*, *desenvolvimento da descrição de circuito* e *verificação*, e o seu sistema de testes faz uso da linguagem *Objetive-C*, permitindo o uso de técnicas escritas de teste de unidade combinadas com técnicas de testes de circuitos digitais.

Palavras-chave: RISC-V. Verilog. Objective-C. FPGA.

Abstract

Elisa Uhura Pereira da Silva. **The Process of Developing a RISC-V CPU**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

RISC-V is an open instruction set architecture available for free for both industry and academic use. Aiming to be the default architecture for all computing devices, it differs from other architectures by the availability of several open implementations designed for use in many kinds of computing tasks. This work presents the development process of a RISC-V processor focusing on the design verification phase. The process has been split into the stages of *planning and preparation*, *circuit description development*, and *verification*. The test system uses the Objective-C language, allowing the use of unit test techniques combined with digital circuit testing techniques.

Keywords: RISC-V. Verilog. Objective-C. FPGA.

Lista de Símbolos

α	Vetor de *bits* equivalente
$<_s$	Comparação se o valor *signed* de um vetor é menor que o de outro
$<_u$	Comparação se o valor *unsigned* de um vetor é menor que o de outro
$\geq_s$	Comparação se o valor *signed* de um vetor é maior ou igual ao de outro
$\geq_u$	Comparação se o valor *unsigned* de um vetor é maior ou igual ao de outro
$\wedge$	Operador binário *and*
$\vee$	Operador binário *or*
$\oplus$	Operador binário *xor*
$\%$	Operador resto
$\ll$	Operador deslocamento para esquerda
$\gg$	Operador deslocamento para direita
$\ggg$	Operador deslocamento aritmético para direita
$\sim$	Operador de inversão de *bits*
$+\!\!+$	Operador de concatenação de vetores
Λ	*Concatenatório*
$\#$	Operador tamanho

Lista de Figuras

Lista de Tabelas

Lista de Programas

Sumário

Capítulo 1

Introdução

1.1 Contextualização

A RISC-V é uma arquitetura de conjunto de instruções (ISA, do inglês *instruction set architecture*) aberta e disponível de forma gratuita, tanto para uso na indústria quanto na academia. Ela é padronizada e patrocinada pela organização internacional sem fins lucrativos *RISC-V International*, que possui dentre seus membros grandes empresas do setor de tecnologia, como *Google*, *Alibaba Cloud* e *Western Digital* (RISC-V, 2021c).

Ela teve início em 2010, através do trabalho do professor Krste Asanović e os alunos de pós-graduação Yunsup Lee e Andrew Waterman no *Parallel Computing Laboratory* da *UC Berkeley* (RISC-V, 2021b), e foi desde o começo, disponível através de uma licença aberta. O projeto evoluiu com o tempo e a ISA passou a incorporar o objetivo principal de ser a arquitetura padrão para todos os dispositivos computacionais (Asanović e Patterson, 2014).

Assim, as decisões de design da arquitetura são tomadas considerando escolhas realizadas por arquiteturas anteriores e suas implicações. Desse modo, a RISC-V evita erros cometidos por arquiteturas anteriores.

Um exemplo dessa decisões é ausência de instruções com *delay slot*, uma técnica usada em arquiteturas, como a MIPS (Etnus, 2003) em que a instrução logo após uma instrução de pulo é executada antes que a primeira seja executada. Esse tipo de decisão de design favorece implementações que fazem uso de técnicas de *pipeline*, populares na época de criação da ISA MIPS, porém aumentam a complexidade de implementações *in-order* da arquitetura.

O diferencial da arquitetura é a disponibilidade de núcleos de propriedade intelectual de semicondutores (IP, do inglês *intellectual property*, termo utilizado para referenciar o design de um circuito digitail) abertos para os mais diversos casos de uso, como o *PicoRV* (Yosys, 2019) voltado para sistemas embarcados, o *Vortex* (Elsabbagh *et al.*, 2020) que usa núcleos RISC-V como aceleradores gráficos, ou o *Xuantie-910* (Chen *et al.*, 2020) destinado ao uso em servidores.

Além da disponibilidade de IPs, diversos programas oferecem suporte a arquitetura,

como o *kernel Linux* desde a versão 4.19, o sistema operacional Debian (Debian, 2021) e diversas linguagens de programação, como *Go* (Pong, 2017), *C/C++* (RISC-V, 2021a) e linguagens que fazem uso da *LLVM* (SiFive, 2021).

1.2 Objetivos

O objetivo deste trabalho foi de apresentar o processo de desenvolvimento de um processador utilizando a arquitetura RISC-V, desde a definição do ambiente até a síntese para uso em uma matriz de porta programáveis (FPGA, do inglês *field-programmable gate array*), com um enfoque na etapa de verificação, através de testes baseados na simulação dos componentes.

1.3 Metodologia

O desenvolvimento de IPs (circuitos digitias) é feito usando uma linguagem de descrição de *hardware* (HDL, do inglês *hardware description language*) sendo a *Verilog* e a *VHDL* as mais utilizadas na indústria. Com base na descrição, é possível simular o comportamento do IP, sintetizar uma configuração para uso em matriz de porta programáveis ou para fabricação de um *chip*.

O desenvolvimento de um processador envolve inúmeras etapas desde sua concepção até a sintetização para uso em uma FPGA ou produção de um *chip*. Além da escolha da ISA, é necessário definir detalhes do ambiente de execução que não são definidos pela arquitetura, escolha do design do processador, desenvolvimento da descrição do *hardware*, uso de testes para verificar a descrição e análises para garantir o comportamento correto do circuito.

Este trabalho foi divido nas etapas de *planejamento e preparação*, *desenvolvimento da descrição* e *verificação*: na etapa de *planejamento e preparação* foram definidas as tecnologias utilizadas, estruturação do projeto e desenvolvimento de bibliotecas auxiliares. Já na etapa de *desenvolvimento da descrição*, foram desenvolvidos os módulos que constituem o processador, e na etapa *verificação*, foram desenvolvidos testes para validar o comportamento do *hardware*.

1.4 Organização do texto

O texto desta monografia é dividido em capítulos que podem apresentar seções e subseções: o Capítulo 2 aborda a notação para manipulação de vetores e conceitos relacionados, a parte utilizada da arquitetura RISC-V, expõe uma introdução às linguagens *Verilog* e *Objective-C* e descreve as partes constituintes de uma FPGA; no Capítulo 3, é descrita a estrutura e *design* do processador e o sistema de testes; o Capítulo 4 apresenta a conclusão da monografia, com uma análise sobre o processo de desenvolvimento da CPU.

Capítulo 2

Desenvolvimento

Este capítulo é iniciado apresentando a notação de vetores de *bits* e conceitos relacionados, e em seguida é introduzida a arquitetura RISC-V, com a apresentação de conceitos da ISA, formato das instruções e a descrição das instruções aceitas. O capítulo encerra-se com uma introdução às linguagens *Verilog* e *Objective-C* e a descrição das partes constituintes de uma FPGA.

2.1 Notação e conceitos

Esta seção apresenta definições e notações que serão usadas ao longo do texto. A notação se faz necessária para garantir uma interpretação clara da descrição do comportamento de instruções e de detalhes de implementação. Os conceitos apresentados possuem definições externas diferentes das apresentadas neste capítulo, pois foram simplificadas para o contexto mais restrito do texto.

2.1.1 Bases

Números prefixados pela letra *b*, *o*, *d* e *h* devem ser interpretados nas bases 2, 8, 10 e 16, respectivamente. A ausência de um prefixo indica que o número deve ser interpretado na base 10. Assim b10 = 2 e hFF = 255. O uso de _ entre dígitos de um número é utilizado para separar regiões e deve ser ignorado na interpretação do número. Por exemplo: 10_780 = 10780.

A notação usual usando um *radix*, como em 10110_2 não foi utilizada para que não colida com a notação utilizada para valores de uma sequência, em que o valor subscrito é utilizado para demarcar a posição na sequência.

2.1.2 Bits

O *bit* é a menor unidade de informação utilizada na computação, podendo assumir dois valores ou estados: 0 e 1. Esses valores também são utilizados para descrever os possíveis estados de um sinal em um circuito lógico.

2.1.3 Vetores

Um vetor é uma sequência de elementos que possuem o mesmo número de estados possíveis, em que seus valores podem ser acessados com base na sua posição no vetor. O primeiro elemento de um vetor começa na posição 0, o segundo na posição 1, e assim por diante. O tamanho de um vetor representa a quantidade de elementos que o mesmo contém, sendo o tamanho sempre maior que 0. Uma posição é considerada válida se ela for maior ou igual a 0 e menor que o tamanho do vetor.

Tanto vetores quanto *bits* são considerados elementos. O tipo de um elemento representa o número de estados possíveis dele e o tipo de um vetor é dado pelo tipo dos elementos na sua sequência elevado pelo tamanho do vetor.

Um vetor de bits A de tamanho B será representado por $A^{1'B}$ ou $A^{'B}$ e possui 2^B possíveis estados. Um vetor C de vetores de bits $A^{'B}$ de tamanho D é representado por $C^{B'D}$ e possui $2^{B \cdot D}$ possíveis estados. O sobrescrito, chamado de formato do vetor, pode ser omitido caso não seja importante ou já seja conhecido.

O valor de um elemento localizado na posição i válida de um vetor A é representado pela notação $A[i]$. O vetor A formado pelos elementos $a_0, a_1, a_2, ..., a_n$, de mesmo tipo é representado por $\{a_0, a_1, a_2, ..., a_n\}$ onde $A[i] = a_i$. Dado dois vetores quaisquer $A^{X'I}$ e $B^{X'J}$, $A \dashv B$ representa um vetor C de tamanho $I + J$ tal que $C[i] = B[i]$ se $i < J$ e $C[i] = A[i - J]$ caso contrário, note que $A \dashv B \neq B \dashv A$.

Dado um vetor A qualquer, $A[m{:}n]$ tal que $m \geq n$ e ambos sejam posições válidas é o vetor dado por $\Lambda_{a=n}^{m}\{A[a]\}$, em que $[m{:}n]$ é denominado intervalo. O operador Λ é análogo ao $\sum$, porém ele aplica operações $\dashv$ no lugar de +. Dado intervalos $a_0, a_1, a_2, ..., a_n$ de um mesmo vetor A, onde cada a_i representa um par $m{:}n$ qualquer, $A[a_0|a_1|a_2|...|a_n] = A[a_0] \dashv A[a_1] \dashv A[a_2] \dashv \ldots \dashv A[a_n]$.

O tamanho de um vetor A é denotado por $\#A$, o tipo de um de seus elementos é denotado por $\#[A]$ e o seu tipo é denotado por $\#(A)$. O valor de um vetor A é denotado por $|A|$ e é dado pela fórmula $\sum_{i=0}^{\#A-1} |A[i]| \cdot \#[A]^i$. O valor de um bit é o seu valor, assim $|0| = 0$ e $|1| = 1$. O valor linha de um vetor A é denotado por $|'A|$ e igual à $\sum_{i=0}^{\#A-1} |'A[\#A - i - 1]| \cdot \#[A]^i$ se $\#[A] > 2$ e $|A|$ caso contrário.

Caso um número a seja usado no lugar de um vetor de bits de formato conhecido, ele representa o vetor cujo valor é igual a a. Por exemplo: $35^{'7} = \{1, 1, 0, 0, 0, 1, 0\} = \text{b10\_0011}^{'7}$.

2.1.4 Extremidade ou endianness

Extremidade ou *endianness* se refere à ordem utilizada para converter ou comparar vetores de mesmo tipo, porém de formatos diferentes. Existem dois tipos de extremidade: *little-endian* e *big-endian*.

Dado dois vetores $A^{I'J}$ e $B^{M'N}$ em que $\#(A) = \#(B)$. B é dito *little-endian* equivalente a A se e somente $|A| = |B|$ e B é dito *big-endian* equivalente a A se e somente $|'A| = |'B|$.

Um exemplo prático para entender o conceito de extremidade é a diferença entre os formatos de um valor armazenado na memória RAM de um computador e o mesmo valor

armazenado no registrador do processador.

Considere o vetor A'^{32} cujo valor é hAFBEEF como o registrador e o vetor $B^{8'4}$ como a região de memória em que o valor do registrador é armazenado. Caso o processador seja *little-endian*, $B^{8'4} = \{\text{hEF}, \text{hBE}, \text{hAFB}, \text{h00}\}$. Caso seja big-endian, $B^{8'4} = \{h00, hAF, hBE, hEF\}$.

Dado um vetor qualquer $A^{M'N}$ e uma endianness E conhecida, αA é um vetor de $M \cdot N$ *bits* E equivalente à A.

2.1.5 Palavra

O termo palavra é usado para designar o tamanho natural dos vetores em um processador ou arquitetura. A palavra costuma determinar o tamanho dos registradores e outros aspectos do processador. Em RISC-V o tamanho da palavra também é designado por XLEN.

2.1.6 Signed e unsigned

O valor de um vetor de *bits* é sempre um número positivo, porém, em certos casos, é desejável uma representação em que o valor do vetor possa assumir um valor negativo. Desse modo, os vetores podem ser tratados como *signed* e *unsigned*, cuja tradução livre seria com e sem sinal.

Quando um vetor de bits A é tratado como *unsigned*, ele pode representar valores de 0 a $\#(A)-1$. E quando ele é tratado como *signed*, ele pode assumir valores de $-\#(A)/2$ a $\#(A)/2-1$. O valor *signed* do vetor, denotado por $|^+A|$, é igual a $(\sum_{i=0}^{\#A-2} A[i] \cdot 2^i) - 2^{\#A-1} \cdot A[\#A-1]$. No caso de um vetor de vetores de *bits* B, $|^+B| = |^+\alpha B|$.

Se tratando de valores *signed* ao definir um vetor de maior tamanho cujo valor *signed* é igual ao de um vetor de menor tamanho, a fim de preservar o valor, os *bits* adicionados devem ser iguais ao valor do *bit* na maior posição do vetor de menor valor. Esse processo é chamado de *sign-extension* ou extensão com sinal, enquanto a *unsigned-extension* ou extensão sem sinal estende o vetor adicionando *bits* com valor 0 nas posições superiores.

2.2 Arquitetura RISC-V

Apesar de parte do nome da ISA conter RISC, um acrônimo para *reduced instruction set computer* (computador com conjunto reduzido de instruções), ela não impede que um processador que implementa a arquitetura possua muitas instruções. Pelo contrário, a arquitetura utiliza uma estratégia de conjunto base de instruções e extensões. Tal estratégia permite que implementações da ISA sejam tão complexas quanto necessário e a ISA seja utilizada em diversos cenários da indústria e da academia.

As especificações oficiais são organizadas e disponibilizadas de forma gratuita no site da *RISC-V International*. Cada parte da especificação possui sua própria versão e *status* de trabalho. Dentre os possíveis *status*, uma parte da especificação pode estar em *draft* (esboço), em que ainda é possível ocorrer grandes alterações em futuras versões, *frozen* (congelada), em que não se esperam grandes alterações em futuras versões, ou *ratified* (ratificada), em que não ocorrerá alterações até uma possível grande revisão.

Toda implementação da ISA deve oferecer um dos conjuntos base e um subconjunto de extensões (oficiais ou não oficiais). Isso permite que implementações para uso em micro controladores não precisem incluir operações com ponto flutuante ou execução privilegiada, que aumentaria o custo do chip, ao mesmo tempo em que ambas as funcionalidades podem ser incluídas em implementações para uso em servidores, onde estas instruções são desejadas.

2.2.1 Harts e EEI

A arquitetura não define todos os detalhes necessários para uma implementação funcional de um processador. Questões como o mapeamento de regiões da memória para dispositivos de entrada e saída, estado inicial do processador ao ser inicializado e outras são definidas pela *execution environment interface* (EEI) ou interface do ambiente de execução. Uma dada EEI possui um ou mais *RISC-V hardware threads* ou *harts*.

Da perspectiva do programa sendo executado em um ambiente de execução, um *hart* é apenas um recurso que automaticamente carrega e executa instruções RISC-V (Waterman e Asanović, 2019a).

Mesmo contendo *hardware* no nome, um *hart* não exige uma implementação física, podendo ser, por exemplo, implementado por emulação via software ou multiplexado, em que vários *harts* a nível de usuário de um sistema operacional são mapeados temporariamente em *harts* físicos para avançarem seu estado.

2.2.2 Formato das instruções

Apesar da ISA poder ser implementada usando tanto o modo *big-endian* quanto *little-endian*, as instruções devem ser armazenadas na memória em palavras de 16 *bits little-endian*. A arquitetura aceita instruções de tamanhos múltiplos de 16 *bits*, sendo seu tamanho padrão 32 *bits*, exceto pelas instruções da extensão de instruções comprimidas, que são de 16 *bits*. Instruções podem ser interpretadas como vetores de *bits*.

Uma instrução a é considerada de 16 *bits* se e somente se a[1:0] for diferente de b11. Uma instrução b é considerada de 32 *bits* se e somente se b[1:0] for igual a b11 e b[4:2] não for igual a b111. Casos em que b[4:2] é igual a b111 são reservados para instruções maiores que fogem do escopo do texto.

Apesar do padrão definir formatos para instruções até 176 *bits* e ter espaço reservado para instruções maiores, a definição delas ainda não foi congelada e pode sofrer alterações no futuro (as definições para 16 e 32 *bits* fazem parte de padrões já retificados). A possibilidade de instruções longas é importante para permitir que pesquisas e produtos que façam uso de *very long word instructions* (Fisher, 1983) possam ser implementados usando RISC-V.

As instruções são divididas em formatos que definem como cada parte das instruções devem ser interpretadas. Com base no tipo, a instrução é subdividida em campos, como rd para representar o registrador de destino. A ISA busca preservar a posição dos campos entre instruções, assim, o campo rd fica na mesma região mesmo em instrução com formatos diferentes, o que simplifica a decodificação da instrução.

As instruções base usam seis formatos diferentes de instruções de 32 *bits* identificados pelas letras R, I, S, B, U e J. Todos possuem o campo $opcode$ de 7 *bits*, localizado no intervalo [6:0] da instrução, que é usado para diferenciar o tipo da instrução. A Tabela 2.1 detalha os campos possíveis e a Tabela 2.2 apresenta os campos presentes em cada formato.

Caso não sejam aceitas instruções cujo tamanho seja um múltiplo ímpar de 16 *bits*, as instruções devem ser alinhadas em vetores de 32 *bits*. Caso haja, as instruções devem ser alinhadas em vetores de 16 *bits*. Considerando a memória um vetor de 8 *bits*, se apenas instruções de 32 *bits* forem aceitas, o endereço da instrução sempre será um múltiplo de 4 e sempre será um múltiplo de 2 caso instruções de 16 *bits* sejam aceitas.

A coluna de função na Tabela 2.2 é utilizada para relacionar descrições de instruções com a sequência de *bits* que ela representa. Os valores entre o primeiro par de parênteses são pré-definidos pela instrução e os valores do segundo par de parênteses são argumentos da instrução. Por exemplo, considere a instrução ADDI, que soma o valor imediato com o valor contido no registrador $rs1$ e guarda o valor no registrador rd, e é descrita por I$(19, 0)(rd, rs1, imm)$, que indica que o campo de $opcode$ possui o valor de 19 e o campo $funct3$ possui o valor de 0 enquanto os campos rd, $rs1$ e imm são argumentos da instrução (onde imm representa o valor do imediato, cuja a instrução utiliza intervalos dele).

Supondo valores para rd, $rs1$ e imm, ADDI$(2, 3, 10)$ = $10'^{32}$[11:0]$+3'^{5}+0'^{3}+2'^{5}+19'^{7}$ = b000000001010_00011_000_00010_0010011$'^{32}$.

Apesar de o imediato ser tratado como um vetor de 32 *bits*, as instruções só podem codificar alguns dos possíveis valores que o imediato pode assumir. Por exemplo, imediatos cujo valor seja ímpar não podem ser representados em instruções que utilizam o formato B.

2.2.3 Formato das instruções comprimidas

As instruções comprimidas fazem uso de instruções de 16 *bits* e adicionam nove formatos de instruções. A Tabela 2.3 detalha os campos possíveis e a Tabela 2.4 detalha os

Campo	# Bits	Intervalo	Definição
$opcode$	7	[6:0]	Determina a instrução ou grupo da instrução e o tipo da instrução.
rd	5	[11:7]	Determina o registrador de destino de uma operação.
$rs1$	5	[19:15]	Determina o registrador usado como primeiro argumento de uma operação.
$rs2$	5	[24:20]	Determina o registrador usado como segundo argumento de uma operação.
$funct3$ $funct7$	3 7	[14:12] [31:25]	Especifica a operação que será realizada pela instrução.
imm[11:0] imm[4:0] imm[11:5] imm[4:1|11:11] imm[12:12|10:5] imm[31:12] imm[20:20|10:1|11:11|19:19|12:12]	12 5 7 5 7 20 20	[31:20] [11:7] [31:25] [11:7] [31:25] [31:12] [31:12]	Define valores de uma região de um valor imediato do tamanho da palavra (XLEN) do conjunto base. Os *bits* não definidos em posições maiores que a maior posição com valor definido na instrução possuem o mesmo valor que o *bit* na maior posição definida. *Bits* não definidos em posições menores que a maior posição com valor definido na instrução possuem o valor de 0.

Tabela 2.1: *Tabela de campos para instruções de 32 bits*

Formato	Campos	Função
R	$funct7 + rs2 + rs1 + funct3 + rd + opcode$	R($opcode, funct3, funct7$)($rd, rs1, rs2$)
I	imm[11:0]$+ rs1 + funct3 + rd + opcode$	I($opcode, funct3$)($rd, rs1, imm$)
S	imm[11:5]$+ rs2 + rs1 + funct3 + imm$[4:0]$+ opcode$	S($opcode, funct3$)($rs1, rs2, imm$)
B	imm[12:12|10:5]$+ rs2 + rs1 + funct3$ $+ imm$[4:1|11:11]$+ opcode$	B($opcode, funct3$)($rs1, rs2, imm$)
U	imm[31:12]$+ rd + opcode$	U($opcode$)(rd, imm)
J	imm[20:20|10:1|11:11|19:19|12:12]$+ rd$ $+ opcode$	J($opcode$)(rd, imm)

Tabela 2.2: *Tabela de formatos para instruções de 32 bits*

campos presentes em cada formato.

Elas fazem parte da extensão C e podem ser mapeadas em instruções equivalentes de 32 *bits*. Devido ao número de *bits* reduzido, alguns formatos fazem uso de campos que apontam para registradores com uma apóstrofe no final que ocupam 3 *bits* ao invés de 5. O valor de um campo rx equivalente a um campo rx' é dado por $rx'^{5} = \mathrm{b01}'^{2} + rx'^{3}$.

Campo	# Bits	Intervalo	Definição
op	2	[1:0]	Determina a instrução ou grupo da instrução e o tipo da instrução.
$rs2$	5	[6:2]	Funciona de forma equivalente aos campos para instruções de 32 *bits*. Campos $rd/rs1$ implicam que o mesmo registrador pode ser utilizado como argumento e ou destino.
$rd/rs1$	5	[11:7]	
rd'	3	[4:2]	
$rs2'$	3	[4:2]	
$rs1'$	3	[9:7]	
$rd'/rs1'$	3	[9:7]	
$funct2$	2	[5:6]	Especifica a operação que será realizada pela instrução.
$funct3$	3	[15:13]	
$funct4$	4	[15:12]	
$funct6$	6	[15:10]	
$immA$	5	[6:2]	Define regiões de um valor imediato de XLEN bits porém diferente dos campos imm de instruções de 32 *bits*, a região representada varia de acordo com a instrução e não com o formato.
$immB$	1	[12:12]	
$immC$	6	[12:7]	
$immD$	8	[12:5]	
$immE$	2	[6:5]	
$immF$	3	[12:10]	
$immG$	11	[12:2]	

Tabela 2.3: *Tabela de campos para instruções de 16 bits*

Formato	Campos	Função
CR	$funct4 + rd/rs1 + rs2 + op$	CR$(op, funct4)(rd/rs1, rs2)$
CI	$funct3 + immB + rd/rs1 + immA + op$	CI$(op, funct3)(rd/rs1, immA, immB)$
CSS	$funct3 + immC + rs2 + op$	CSS$(op, funct3)(rs2, immC)$
CIW	$funct3 + immD + rd' + op$	CIW$(op, funct3)(rd', immD)$
CL	$funct3 + immF + rs1' + immE + rd' + op$	CL$(op, funct3)(rd', rs1', immE, immF)$
CS	$funct3 + immF + rs1' + immE + rs2' + op$	CS$(op, funct3)(rs1', rs2', immE, immF)$
CA	$funct6 + rd'/rs1' + funct2 + rs2' + op$	CA$(op, funct2, funct6)(rd'/rs1', rs2')$
CB	$funct3 + immF + rs1' + immA + op$	CB$(op, funct3)(rs1', immA, immF)$
CJ	$funct3 + immG + op$	CJ$(op, funct3)(immG)$

Tabela 2.4: *Tabela de formatos para instruções de 16 bits*

As funções descritas para os formatos comprimidos não possuem informações o suficiente para definir como os intervalos do valor imediato são registrados na instrução. Assim, a descrição das funções deixa explícito o mapeamento através de comentários após a função. Por exemplo, a instrução C.LI(rd, imm'^{6}) é descrita por CI$(1, 2)(rd, imm[4:0], imm[5:5])$,

em que rd é diferente de 0. C.LI altera o valor contido no registrador rd para ser igual ao valor do imm.

Supondo valores para rd e imm, C.LI$(2, 27)$ = $2'^{3} \# 27'^{32}[5{:}5] \# 2'^{5} \# 27'^{32}[4{:}0] \# 2'^{2}$ = b010_0_00010_11011_10$'^{16}$.

2.2.4 Processamento da instrução

Instruções descrevem alterações no estado de um *hart*, sem especificar detalhadamente quando e como o estado é alterado. Definida uma sequência de instruções, o estado *real* do *hart* entre execuções de instruções não precisa refletir o especificado pelas instruções, e nem mesmo induz que as instruções sejam executadas uma após a outra. Isso permite que diferentes técnicas sejam utilizadas para implementar um processador RISC-V.

A descrição de uma instrução de soma como "o valor do registrador A passa a ser igual a soma do valor do registrador B com o valor do registrador C", não implica necessariamente que após sua execução o valor do registrador A seja o da soma, e sim que caso o seu valor seja observado por alguma outra execução de instrução sem que seu valor tenha sido alterado por instruções intermediários, o valor observado deve refletir a soma.

2.2.5 Instruções base

Os conjuntos de instruções base incluem instruções relacionadas à leitura e à escrita de inteiros, computação com inteiros, transferência de controle, ordenação de memória e chamadas para o ambiente de execução.

Atualmente a RISC-V engloba quatro conjuntos base, três conjuntos com trinta e dois registradores, RV32I, RV64I e RV128I com palavras de 32, 64 e 128 *bits* respectivamente e um conjunto de dezesseis registradores com palavras de 32 *bits* RV32E. Os registradores são identificados por $x0$ a $x31$ (o RV32E só possui até o registrador $x15$).

A maioria das instruções são compartilhadas entre os conjuntos base, com exceção das instruções relacionadas a palavras de 64 *bits*, que só estão disponíveis nos conjuntos RV64I e RV128I, e as instruções relacionadas a palavras de 128 *bits*, que só estão disponíveis no conjunto RV128I.

O RV32E é similar ao RV32I, exceto pelo fato de que as instruções que almejam os registradores que não existem no RV32E passam a ser consideradas livres para uso por extensões.

Como o núcleo implementado usa apenas o conjunto RV32I, não serão apresentados detalhes dos outros conjuntos base. Porém, é interessante notar que a arquitetura provê mecanismos para que um mesmo *hart* implemente mais de um conjunto base e alterne entre eles (Waterman e Asanović, 2019b).

Registradores

A arquitetura define que cada implementação deve possuir trinta e dois registradores de XLEN *bits* onde $x0$ sempre possui o valor 0 e descarta qualquer escrita e XLEN é o tamanho definido para os registradores. Além dos registradores de uso geral, existe um

registrador de XLEN *bits* denominado *pc* (do inglês *program counter*, com tradução livre contador de programa) que aponta para o endereço de memória em que a instrução que está sendo executada se localiza.

Memória e instruções de leitura e escrita

Para um *hart*, a memória é tratada como um único espaço circular de 2^{XLEN} bytes. O modelo padrão de consistência de memória é o *RISC-V Weak Memory Ordering* (RVWMO) (Waterman e Asanović, 2019a) e o *endianness* é definido pela EEI.

Uma leitura de 4 *bytes* a partir de $2^{\text{XLEN}} - 2$ equivale ao intervalo $[2^{\text{XLEN}} + 1{:}2^{\text{XLEN}} - 2]$, cujas posições são computadas considerando seu valor módulo 2^{XLEN}. Considerando $mem^{8'2^{\text{XLEN}}}$ o vetor da memória, $mem[2^{\text{XLEN}} + 1{:}2^{\text{XLEN}} - 2] = mem[2^{\text{XLEN}} - 2], mem[2^{\text{XLEN}} - 1], mem[0], mem[1]$.

A arquitetura usa um modelo de *load-store*, onde valores podem ser lidos ou escritos entre registradores e memória, porém operações como soma ou multiplicação utilizam apenas valores contidos nos registradores.

O RV32I aceita operações de leitura e escrita de valores de 8, 16 e 32 *bits*. Por padrão, os valores são tratados como *signed* e operações de leitura de 8 e 16 *bits* possuem variantes *unsigned* que não realizam *signed-extension* ao definir o valor do registrador.

As instruções de leitura utilizam o formato I em que o valor contido no registrador $rs1$ é somado com o valor do imediato para computar um endereço que é utilizado para ler uma quantidade de *bytes* a serem escritos no registrador rd.

As instruções de escrita utilizam o formato S em que o valor contido no registrador $rs1$ é somado com o valor do imediato para computar um endereço utilizado para alterar o valor de certos *bytes* da memória com base no valor contido no registrador $rs2$.

Considerando $mem^{8'2^{\text{XLEN}}}$ o vetor da memória e $reg^{32'32}$ o vetor de registradores, a Tabela 2.5 descreve as instruções de leitura e escrita.

Instruções de operação entre registradores e ou imediatos

O conjunto base de instruções permite realizar uma gama de operações entre vetores de *bits*. As operações podem ser computadas utilizando o valor de dois registradores ou um registrador e um valor imediato derivado da instrução e o resultado é armazenado em algum registrador.

A arquitetura inclui as operações de soma (+), subtração (−), comparação de menor *signed* ($<_s$) e *unsigned* $<_u$, operações binárias *and* ($\wedge$), *or* ($\vee$) e *xor* ($\oplus$) e operações de deslocamento para esquerda ($\ll$), para direita ($\gg$) e aritmético para direita ($\ggg$).

Sendo A e B dois vetores de *bits* de mesmo tamanho x:

- $A + B = (|A| + |B| \% 2^x)'^x$
- $A - B = A + (-B)$

Mnemônico	Argumentos	Definição	Descrição
LB	$rd, rs1, imm$	I$(3, 0)(rd, rs1, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \|^{+}reg[\|rd\|]\|$ passa a ser igual a $\|^{+}\alpha mem[a{:}a]\|$
LH	$rd, rs1, imm$	I$(3, 1)(rd, rs1, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \|^{+}reg[\|rd\|]\|$ passa a ser igual a $\|^{+}\alpha mem[a+1{:}a]\|$
LW	$rd, rs1, imm$	I$(3, 2)(rd, rs1, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \|^{+}reg[\|rd\|]\|$ passa a ser igual a $\|^{+}\alpha mem[a+3{:}a]\|$
LBU	$rd, rs1, imm$	I$(3, 4)(rd, rs1, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \|reg[\|rd\|]\|$ passa a ser igual a $\|\alpha mem[a{:}a]\|$
LHU	$rd, rs1, imm$	I$(3, 5)(rd, rs1, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \|reg[\|rd\|]\|$ passa a ser igual a $\|\alpha mem[a+1{:}a]\|$
SB	$rs1, rs2, imm$	S$(43, 0)(rs1, rs2, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \alpha mem[a{:}a]$ passa a ser igual a $reg[\|rs2\|][7:0]$
SH	$rs1, rs2, imm$	S$(43, 1)(rs1, rs2, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \alpha mem[a+1{:}a]$ passa a ser igual a $reg[\|rs2\|][15:0]$
SW	$rs1, rs2, imm$	S$(43, 2)(rs1, rs2, imm)$	Considerando o endereço $a = \|reg[\|rs1\|]\| + \|imm\|, \alpha mem[a+3{:}a]$ passa a ser igual a $reg[\|rs2\|]$

Tabela 2.5: *Tabela de instruções de leitura e escrita*

- $A <_s B = \begin{cases} 1'^{x}, & \text{se } |^{+}A| < |^{+}B| \\ 0'^{x}, & \text{caso contrário} \end{cases}$
- $A <_u B = \begin{cases} 1'^{x}, & \text{se } |A| < |B| \\ 0'^{x}, & \text{caso contrário} \end{cases}$
- $A \wedge B = \Lambda_{i=1}^{x}\{A[x-i] \wedge B[x-i]\}$
- $A \vee B = \Lambda_{i=1}^{x}\{A[x-i] \vee B[x-i]\}$
- $A \oplus B = \Lambda_{i=1}^{x}\{A[x-i] \oplus B[x-i]\}$
- $A \ll B = \Lambda_{i=1}^{x} \begin{cases} \{A[x-i+\|B[4{:}0]\|]\}, & \text{se } x-i+\|B[4{:}0]\| \text{ for uma posição válida} \\ 0'^{1}, & \text{caso contrário} \end{cases}$
- $A \gg B = \Lambda_{i=1}^{x} \begin{cases} \{A[x-i-\|B[4{:}0]\|]\}, & \text{se } x-i-\|B[4{:}0]\| \text{ for uma posição válida} \\ 0'^{1}, & \text{caso contrário} \end{cases}$

- $A \ggg B = \Lambda_{i=1}^{x} \begin{cases} \{A[x-i-|B[4{:}0]|]\}, & \text{se } x-i-|B[4{:}0]| \text{ for uma posição válida} \\ \{A[x-1]\}, & \text{caso contrário} \end{cases}$

Em que % é o operador de resto: dados $a, b, c, d \in \mathbb{N}$, $a/b = c \cdot b + d$: $a\%b := d$. $-B$ é o complemento para dois de B dado por $B \oplus (2^x - 1)'^x + 1'^x$.

Instruções entre registradores e imediatos utilizam o formato I, enquanto operações entre registradores utilizam o formato R.

A arquitetura oferece duas instruções que utilizam o formato U para construção de constantes e cálculo de endereços relativos ao *pc*. A LUI, quando utilizada em conjunto com a ADDI, permite que qualquer valor de 32 *bits* seja escrito em um registrador, e a instrução AUIPC permite copiar o valor do *pc* para um dos registrados de uso geral.

Considerando $reg'^{32}{}^{32}$ o vetor de registradores e pc'^{32}, a Tabela 2.6 descreve as instruções de operações entre registradores e/ou imediatos.

Instruções de transferência de controle

Com base no valor do *pc*, o *hart* carrega a instrução localizada no endereço de memória e a executa. Caso a instrução não afete o valor do *pc*, ele é incrementado pelo número de *bytes* da instrução e o *hart* carrega e executa a próxima instrução.

Instruções de pulo utilizam o formato J e I e são utilizadas para alterar o *pc*, alterando a próxima instrução a ser processada. O endereço da próxima instrução que seria executada caso o pulo não tivesse ocorrido é armazenado em *rd*, processo chamado de *link*.

Instruções de ramificação utilizam o formato B e alteram o valor do *pc* de forma condicional. Caso o critério de teste da instrução seja atendido, ele altera o valor do *pc* com base no valor imediato. Caso contrário, o *pc* é incrementado como nas outras instruções.

Considerando $reg'^{32}{}^{32}$ o vetor de registradores e pc'^{32}, a Tabela 2.7 descreve as instruções de transferência de controle. Os operadores $\geq_u$ e $\geq_s$ comparam se os valores *unsigned* e *signed* do vetor da esquerda são maiores ou iguais aos respectivos valores do vetor da direita.

Caso a implementação não inclua instruções cujo tamanho seja um múltiplo ímpar de 16 *bits* e a instrução executada altere o valor do *pc* para um endereço que não seja alinhada em 32 *bits*, uma exceção de *instruction-address-misaligned* (endereço de instrução desalinhado) deve ser levantada.

Como o valor do *pc* sempre é par antes de executar uma instrução e é garantido que, após a execução de uma instrução de transferência de controle, o valor do *pc* continue par, seja pelos valores legais para o imediato no formato da instrução ou pela descrição da instrução (JALR garante $pc[0] = 0$). Já em implementações que aceitam instruções cujo tamanho seja um múltiplo ímpar de 16 *bits*, não é possível que a exceção de endereço desalinhado seja levantada.

Mnemônico	Argumentos	Definição	Descrição
ADDI	$rd, rs1, imm$	I$(19, 0)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] + imm$
SLTI	$rd, rs1, imm$	I$(19, 2)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] <_s imm$
SLTIU	$rd, rs1, imm$	I$(19, 3)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] <_u imm$
XORI	$rd, rs1, imm$	I$(19, 4)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \oplus imm$
ORI	$rd, rs1, imm$	I$(19, 6)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \vee imm$
ANDI	$rd, rs1, imm$	I$(19, 7)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \wedge imm$
SLLI	$rd, rs1, imm$	I$(19, 1)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \ll imm$ e $\|imm\| < 2^5$
SRLI	$rd, rs1, imm$	I$(19, 5)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \gg imm$ e $\|imm\| < 2^5$
SRAI	$rd, rs1, imm$	I$(19, 5)(rd, rs1, imm)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \ggg imm$ e $2^{10} \leq \|imm\| < 2^{10} + 2^5$
ADD	$rd, rs1, rs2$	R$(51, 0, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] + reg[\|rs2\|]$
SUB	$rd, rs1, rs2$	R$(51, 0, 32)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] - reg[\|rs2\|]$
SLL	$rd, rs1, rs2$	R$(51, 1, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \ll reg[\|rs2\|]$
SLT	$rd, rs1, rs2$	R$(51, 2, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] <_s reg[\|rs2\|]$
SLTU	$rd, rs1, rs2$	R$(51, 3, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] <_u reg[\|rs2\|]$
XOR	$rd, rs1, rs2$	R$(51, 4, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \oplus reg[\|rs2\|]$
SRL	$rd, rs1, rs2$	R$(51, 5, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \gg reg[\|rs2\|]$
SRA	$rd, rs1, rs2$	R$(51, 5, 32)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \ggg reg[\|rs2\|]$
OR	$rd, rs1, rs2$	R$(51, 6, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \vee reg[\|rs2\|]$
AND	$rd, rs1, rs2$	R$(51, 7, 0)(rd, rs1, rs2)$	$reg[\|rd\|]$ passa a ser igual a $reg[\|rs1\|] \wedge reg[\|rs2\|]$
LUI	rd, imm	U$(55)(rd, imm)$	$reg[\|rd\|]$ passa a ser igual a $imm[31{:}12] + 0'^{12}$
AUIPC	rd, imm	U$(55)(rd, imm)$	$reg[\|rd\|]$ passa a ser igual a $imm + pc$

Tabela 2.6: *Tabela de instruções de operações entre registradores e/ou imediatos*

Mnemônico	Argumentos	Definição	Descrição
JAL	rd, imm	J(111)(rd, imm)	pc passa a ser igual a $pc + imm$ e $reg[\|rd\|]$ passa a ser igual a $pc + 4'^{32}$
JALR	$rd, rs1, imm$	I(103, 0)($rd, rs1, imm$)	pc passa a ser igual a $(reg[\|rs1\|] + imm)[31:1] + 0'^{1}$ e $reg[\|rd\|]$ passa a ser igual a $pc + 4'^{32}$
BEQ	$rs1, rs2, imm$	B(99, 0)($rs1, rs2, imm$)	Se $rs1 = rs2$, pc passa a ser igual a $pc+imm$. Caso contrário pc passa a ser igual a $pc + 4'^{32}$
BNE	$rs1, rs2, imm$	B(99, 1)($rs1, rs2, imm$)	Se $rs1 \neq rs2$, pc passa a ser igual a $pc+imm$. Caso contrário pc passa a ser igual a $pc + 4'^{32}$
BLT	$rs1, rs2, imm$	B(99, 4)($rs1, rs2, imm$)	Se $rs1 <_s rs2$, pc passa a ser igual a $pc + imm$. Caso contrário pc passa a ser igual a $pc + 4'^{32}$
BGE	$rs1, rs2, imm$	B(99, 5)($rs1, rs2, imm$)	Se $rs1 \geq_s rs2$, pc passa a ser igual a $pc + imm$. Caso contrário pc passa a ser igual a $pc + 4'^{32}$
BLTU	$rs1, rs2, imm$	B(99, 6)($rs1, rs2, imm$)	Se $rs1 <_u rs2$, pc passa a ser igual a $pc + imm$. Caso contrário pc passa a ser igual a $pc + 4'^{32}$
BGEU	$rs1, rs2, imm$	B(99, 6)($rs1, rs2, imm$)	Se $rs1 \geq_u rs2$, pc passa a ser igual a $pc + imm$. Caso contrário pc passa a ser igual a $pc + 4'^{32}$

Tabela 2.7: *Tabela de instruções de transferência de controle*

Instruções de barreira e chamadas de ambiente

A arquitetura utiliza um modelo de ordenação fraca que permite que a ordem das operações de leitura e escrita sejam alteradas desde que mantenham sua ordem perante a perspectiva do *hart* que as executou. Para garantir que essas operações sejam observadas na ordem desejada por outros *harts*, é disponibilizada a instrução FENCE.

A instrução FENCE utiliza o formato I e, na versão atual da especificação, os valores dos registradores passados são ignorados. Os 11 *bits* do imediato são interpretados utilizando uma estrutura específica. Considerando imm'^{32} o vetor do imediato, a instrução possui os seguintes campos:

- $imm[0]$: Marcador de escrita no conjunto sucessor (SW).
- $imm[1]$: Marcador de leitura no conjunto sucessor (SR).
- $imm[2]$: Marcador de saída no conjunto sucessor (SO).
- $imm[3]$: Marcador de entrada no conjunto sucessor (SI).
- $imm[4]$: Marcador de escrita no conjunto predecessor (PW).

- $imm[5]$: Marcador de leitura no conjunto predecessor (PR).
- $imm[6]$: Marcador de saída no conjunto predecessor (PO).
- $imm[7]$: Marcador de entrada no conjunto predecessor (PI).
- $imm[11:8]$: Modo de barreira ou *(*fence mode) (fm).

O valor de fm determina como a barreira deve ser interpretada. Caso $|fm| = 0$, qualquer operação do tipo marcado no conjunto predecessor que tenha sido executada até o momento da instrução de barreira não pode ser observada por outros *harts* depois de qualquer operação do tipo marcado no conjunto sucessor executada depois da instrução de barreira. Caso $|fm| = 1$ e apenas PR, PW, SR, SW estejam marcados, operações de escrita do conjunto sucessor pode ocorrer antes de operações de leitura do conjunto predecessor. Esse modo com $|fm| = 1$ é nomeado *total store order* (TSO) e sua implementação é opcional.

O sistema oferece as instruções ECALL e EBREAK para realizarem *traps* para o ambiente de execução. ECALL é utilizada para solicitar um serviço do ambiente, e EBREAK é utilizada para devolver o controle para o *debugger*. Elas utilizam o formato I.

Para alterar o comportamento do *hart*, é utilizado um conjunto de instruções que manipulam os registradores de controle e *status* (*control and status registers*, ou CSRs) que não fazem parte dos conjuntos base. A Tabela 2.8 descreve as instruções de barreira e chamada de ambiente.

Mnemônico	Argumentos	Definição	Descrição
FENCE	$(rd, rs1, imm)$	I$(15, 0)(rd, rs1, imm)$	Realiza a operação de barreira conforme descrito na seção 2.2.5
ECALL	()	I$(115, 0)(0, 0, 0)$	Levanta uma *trap* de chamada para o ambiente
EBREAK	()	I$(115, 0)(0, 0, 1)$	Devolve o controle para o ambiente de *debug*

Tabela 2.8: *Tabela de instruções de barreira e chamada de ambiente*

Instruções que não alteram o estado

A arquitetura padrão define instruções que não alteram o estado como dicas. Assim, a operação SUB(0, 2, 3) é reservada para representar informações extras sobre o contexto de execução. Apesar da classificação de instruções como dicas, elas ainda não possuem uso na especificação.

A instrução ADDI$(0, 0, 0)$ é reservada como a instrução NOP (*no operation*) e não é considerada uma instrução de dica. Um exemplo de uso de instruções que não alteram o estado para alterar o comportamento presente em Waterman e Asanović, 2019a, é a sequência de instruções que permitem *semihosting*: {SLLI$(0, 0,$ h1f), EBREAK, SRAI$(0, 0, 7)$}.

Quando essa sequência de instruções aparece em um processador que implementa a funcionalidade, ao invés de executar a instrução de BREAK para devolver o controle para o *debugger*, ela é executada como uma ECALL para o ambiente de *debug*.

2.2.6 Registradores de controle e *status*

A RISC-V define um espaço separado de 4096 CSRs para cada *hart*. Cada CSR possui uma funcionalidade específica, como contador de ticks, vetor para tratamento de exceção e informação sobre as extensões implementadas. A arquitetura trata esse espaço separado do espaço de memória e a operação de FENCE não os afetam.

O comportamento ao ler ou escrever um CSR é complexo e depende de qual CSR é utilizado. As instruções podem ou não gerar eventos de leitura e/ou escrita dependendo dos argumentos, e cada CSR possui um comportamento específico em relação a leitura e escrita de valores:

- *Reserved Writes Preserve Value, Read Ignores Values* (WPRI), ou em tradução livre, escritas reservadas preservam valor, leitura ignora valor: Alguns campos de um CSR podem estar reservados para uso futuro, assim a leitura desses campos deve ser ignorada e é necessário preservar (não alterar) o valor desses campos em caso de escritas do CSR.
- *Write/Read Only Legal Values* (WLRI), ou em tradução livre, escrita/leitura apenas de valores válidos: Apenas valores válidos podem ser escritos no CSR e não podem assumir que um valor válido será lido antes de escrevê-lo. É possível que seja levantada uma exceção de instrução ilegal caso o valor a ser escrito no CSR seja inválido, porém esse comportamento é opcional.
- *Write Any Value, Reads Legal Values* (WARL), ou em tradução livre, escreve qualquer valor, lê valores válidos: A escrita de valores inválidos não deve levantar exceções e a leitura é sempre de um valor válido. Isso permite que a escrita seja usada para validar se alguma funcionalidade está disponível e em caso de sucesso o valor atualizado será lido.

A largura de um CSR em conjuntos base maiores que 32 *bits* pode ter seu valor alterado de forma dinâmica, porém esse comportamento não será abordado aqui, uma vez que, para os conjuntos de 32 *bits*, o tamanho é fixado em 32 *bits*.

A Tabela 2.9 lista as instruções para manipulação de CSRs que definem a extensão Zicsr. Considerando $reg^{32'32}$ o vetor de registradores e $csr^{32'4096}$ o espaço de CSRs para a descrição das instruções:

- CSRRW escreve o valor de $reg[|rs1|]$ em $csr[|imm[11:0]|]$ e o valor antigo de $csr[|imm[11:0]|]$ é armazenado em $reg[|rd|]$. Caso $rd = 0'^5$, a operação de leitura não ocorre.
- CSRRS escreve o valor de $csr[|imm[11:0]|] \vee reg[|rs1|]$ em $csr[|imm[11:0]|]$ e o valor antigo de $csr[|imm[11:0]|]$ é armazenado em $reg[|rd|]$. Caso $rs1 = 0'^5$ a operação de escrita não ocorre.
- CSRRC escreve o valor de $csr[|imm[11:0]|] \wedge \sim reg[|rs1|]$ em $csr[|imm[11:0]|]$ e o valor antigo de $csr[|imm[11:0]|]$ é armazenado em $reg[|rd|]$. Caso $rs1 = 0'^5$ a operação de escrita não ocorre. Onde operador ~ inverte os *bits* do vetor prefixado por ele.
- CSRRWI escreve o valor de $|uimm|'^{32}$ em $csr[|imm[11:0]|]$ e o valor antigo de $csr[|imm[11:0]|]$ é armazenado em $reg[|rd|]$. Caso $rd = 0'^5$, a operação de leitura

não ocorre.

- CSRRSI escreve o valor de $csr[|imm[11:0]|] \vee |uimm|^{'32}$ em $csr[|imm[11:0]|]$ e o valor antigo de $csr[|imm[11:0]|]$ é armazenado em $reg[|rd|]$. Caso $uimm = 0^{'5}$ a operação de escrita não ocorre.
- CSRRCI escreve o valor de $csr[|imm[11:0]|] \wedge \sim |uimm|^{'32}$ em $csr[|imm[11:0]|]$ e o valor antigo de $csr[|imm[11:0]|]$ é armazenado em $reg[|rd|]$. Caso $uimm = 0^{'5}$ a operação de escrita não ocorre.

Mnemônico	Argumentos	Definição
CSRRW	$(rd, rs1, imm)$	I$(115, 1)(rd, rs1, imm)$
CSRRS	$(rd, rs1, imm)$	I$(115, 2)(rd, rs1, imm)$
CSRRC	$(rd, rs1, imm)$	I$(115, 3)(rd, rs1, imm)$
CSRRWI	$(rd, uimm, imm)$	I$(115, 5)(rd, uimm, imm)$
CSRRSI	$(rd, uimm, imm)$	I$(115, 6)(rd, uimm, imm)$
CSRRCI	$(rd, uimm, imm)$	I$(115, 7)(rd, uimm, imm)$

Tabela 2.9: *Tabela de instruções para manipulação de CSRs*

O conjunto base especifica um conjunto de contadores opcionais de 64 *bits* mapeados em pares de CSRs. Os principais contadores são RDCYCLE, RDTIME e RDINSTRET que contam respectivamente o número de ciclos de *clock*, um número incrementado em um intervalo de tempo constante determinado pela EEI e o número de instruções executadas. As partes inferiores ([31:0]) desses contadores estão localizadas nas posições hC00, hC01 e hC02 e as partes superiores ([63:32]) nas posições hC80, hC81 e hC82, e são apenas para leitura.

2.2.7 Arquitetura privilegiada e tratamento de exceções

A especificação da parte privilegiada da arquitetura, Waterman e Asanović, 2019b, descreve o comportamento relacionado à execução privilegiada, memória virtual, tratamento de exceções, entre outros. A implementação da parte privilegiada pode ser apenas parcial e variar de forma extensiva entre diversas implementações RISC-V, porém, é esperado que as partes em comum implementadas por sistemas diferentes apresentem o mesmo comportamento.

Três níveis de privilégios são definidos: usuário (U), supervisor (S) e máquina (M). Os modos são codificados por $0^{'2}$ para usuário, $1^{'2}$ para supervisor e $3^{'2}$ para máquina. Toda implementação deve incluir o modo máquina e o modo usuário e supervisor são opcionais, sendo obrigatória a inclusão do modo usuário caso o supervisor seja incluído.

Por brevidade só será abordado o mecanismo mínimo para lidar com exceções a nível de máquina. Ele é implementado através dos CSRs *mstatus*, *mtvec*, *mscratch*, *mepc*, *mcause* e *mtval* localizados nas posições h300, h305, h340, h341, h342 e h343 respectivamente.

Considerando $csr^{32'4096}$ o espaço de CSRs, $mstatus = csr$[h300], $mtvec = csr$[h305], $mscratch = csr$[h340], $mepc = csr$[h341], $mcause = csr$[h342] e $mtval = csr$[h343], a

listagem a seguir apresenta os campos de interesse:

- $mstatus$[3:3] (MIE) determina se interrupções estão habilitadas caso seu valor seja 1 e 0 caso contrário.
- $mstatus$[7:7] (MPIE) guarda o MIE anterior e é utilizada para restaurar o MIE após a execução da instrução MRET.
- $mstatus$[12:11] (MPP) determina o nível de privilégio a ser definido quando a instrução para retorno de interrupção ou exceção ocorra.
- $mtvec$[1:0] (MODE) determina como o endereço em caso de interrupções é utilizado. Caso o valor seja 0, o endereço é utilizado diretamente. Caso o valor seja 1, o endereço utilizado é somado com 4 vezes o valor da causa antes de ser atribuído ao pc. Outros valores foram reservados para uso futuro.
- $mtvec$[31:2] (BASE) guarda o endereço utilizado em caso de interrupções. Ele é computado por $mtvec[31:2] + 0^{'2}$.
- $mscratch$ é utilizado para armazenar um valor no modo de máquina. Ele pode ser utilizado para computar o contexto a ser carregado no caso de uma interrupção.
- $mepc$ guarda o valor do pc antes da interrupção acontecer. Ele é utilizado quando a instrução de retorno de interrupção é executada para definir o valor do pc.
- $mcause$ guarda a causa da interrupção com o *bit* da maior posição sendo utilizado para demarcar se foi uma interrupção em si, e o restante dos *bits* guardam o valor da causa. A Tabela 2.10 lista as causas de interesse.
- $mtval$ é utilizado para armazenar valores relacionados à interrupção como a instrução que causou uma exceção de instrução ilegal.

Quando ocorre uma interrupção, MPIE guarda o valor de MIE, MIE passa a valer 0, o MPP passa a valer o nível de execução no momento em que a interrupção ocorreu, $mepc$ recebe o valor do pc e os campos $mcause$ e $mtval$ são preenchidos de acordo. O valor do pc é atualizado de acordo com o valor de $mtvec$ e o *hart* continua a processar as instruções no modo de máquina.

A instrução MRET é utilizada para retornar da resolução de uma exceção. Ela utiliza o formato I e possui a definição $I(115, 0)(0, 0, 700)$. Quando ela é executada, pc recebe o valor de $mepc$, MIE recebe o valor de MPIE, MPIE passa a valer 1 e MPP é redefinido para o valor do modo de usuário (ou de máquina caso o modo de usuário não esteja disponível) e o *hart* continua a processar as instruções com o modo de acordo com o antigo valor de MPP.

O sistema de temporizador utiliza dois campos de 64 *bits*, $mtime$ e $mtimecmp$. O $mtime$ é acessado pelo RDTIME enquanto o $mtimecmp$ é mapeado na memória principal pelo EEI. Enquanto o $mtime$ for maior que o $mtimecmp$, a interrupção de máquina por temporizador é emitida.

Interrupção	Código da Exceção	Descrição
1	7	Interrupção de máquina por temporizador
1	11	Interrupção de máquina por uma causa externa
0	2	Instrução ilegal
0	8	Chamada de ambiente do nível de usuário
0	10	Chamada de ambiente do nível de máquina

Tabela 2.10: *Tabela de causas de interrupções*

2.2.8 Instruções comprimidas

A extensão C é constituída por instruções comprimidas de 16 *bits*. Elas são consideradas comprimidas pois é possível mapear cada uma delas em instruções de 32 *bits*. Devido ao menor espaço de instruções válidas ($2^{16} - 2^{14}$), algumas das instruções se sobrepõem e a instrução a ser executada é determinada pelo conjunto base. Instruções da extensão C não disponíveis para o conjunto base RV32I não serão cobertas nesta seção.

A função da extensão é permitir a redução do tamanho de executáveis, com uma redução estimada de 20-30% no tamanho da seção de texto dos programas. A instrução $0^{'16}$ é reservada como instrução ilegal. A Tabela 2.11 lista as instruções comprimidas que definem a extensão C e são aceitas por implementações que utilizam a base RV32I. Considerando $reg^{32'32}$ o vetor de registradores e $mem^{8'2^{XLEN}}$ o vetor da memória:

- C.LWSP($rd, uimm$) expande para LW($|rd|, 2, |uimm|$).
- C.SWSP($rs2, uimm$) expande para SW($2, |rs2|, |uimm|$).
- C.LW($rd', rs1', uimm$) expande para LW($|rd|, |rs1|, |uimm|$).
- C.SW($rs1', rs2', uimm$) expande para SW($|rs1|, |rs2|, |uimm|$).
- C.J(imm) expande para JAL($0, ||^{+}imm|^{'32}|$).
- C.JAL(imm) expande para JAL($1, ||^{+}imm|^{'32}|$), note que a próxima instrução está em $pc + 2$.
- C.JR($rs1$) expande para JALR($0, rs1, 0$), note que a próxima instrução está em $pc + 2$.
- C.JALR($rs1$) expande para JALR($1, rs1, 0$), note que a próxima instrução está em $pc + 2$.
- C.BEQZ($rs1', imm$) expande para BEQ($|rs1|, 0, ||^{+}imm|^{'32}|$).
- C.BNEZ($rs1', imm$) expande para BNE($|rs1|, 0, ||^{+}imm|^{'32}|$).
- C.LI(rd, imm) expande para ADDI($|rd|, 0, ||^{+}imm|^{'32}|$).
- C.LUI($rd, nzimm$) expande para LUI($|rd|, ||^{+}nzimm|^{'32}|$).
- C.ADDI($rs1/rd, nzimm$) expande para ADDI($|rs1/rd|, |rs1/rd|, ||^{+}nzimm|^{'32}|$).
- C.ADDI16SP($nzimmm$) expande para ADDI($2, 2, ||^{+}nzimm|^{'32}|$).

- C.ADDI4SPN(rd', $nzuimm$) expande para ADDI($|rd|$, 2, $||^{+}nzimm|^{'32}|$).
- C.SLLI($rs1/rd$, $nzuimm$) expande para SLLI($|rs1/rd|$, $|rs1/rd|$, $|nzuimm|$).
- C.SRLI($rs1'/rd'$, $uimm$) expande para SRLI($|rs1/rd|$, $|rs1/rd|$, $|uimm|$).
- C.SRAI($rs1'/rd'$, $uimm$) expande para SRAI($|rs1/rd|$, $|rs1/rd|$, $|uimm|$).
- C.ANDI($rs1'/rd'$, imm) expande para ANDI($|rs1/rd|$, $|rs1/rd|$, $||^{+}imm|^{'32}|$).
- C.MV(rd, $rs2$) expande para ADD($|rd|$, 0, $|rs2|$).
- C.ADD($rs1$, $rs2$) expande para ADD($|rs1|$, $|rs1|$, $|rs2|$).
- C.AND($rs1'/rd'$, $rs2'$) expande para AND($|rs1/rd|$, $|rs1/rd|$, $|rs2|$).
- C.OR($rs1'/rd'$, $rs2'$) expande para OR($|rs1/rd|$, $|rs1/rd|$, $|rs2|$).
- C.XOR($rs1'/rd'$, $rs2'$) expande para XOR($|rs1/rd|$, $|rs1/rd|$, $|rs2|$).
- C.SUB($rs1'/rd'$, $rs2'$) expande para SUB($|rs1/rd|$, $|rs1/rd|$, $|rs2|$).
- C.NOP expande para NOP, definida como ADDI(0, 0, 0).
- C.EBREAK expande para EBREAK().

2.2.9 Nomenclatura

A arquitetura define um esquema de nomenclatura para definir quais extensões um *hart* RISC-V aceita. A nomenclatura não é sensível a caixa alta. De modo geral, a primeira parte é composta pela arquitetura base, seguida pelas extensões oficiais implementadas, que são representadas por uma letra do alfabeto ou prefixadas pela letra Z.

A letra G é usada como uma abreviação para as extensões IMAFDZicsr_Zifencei. Extensões que são consideradas dependências de outras podem ser omitidas, como é o caso das extensões F e Zicsr que são dependências da extensão D. O uso de _ é apenas para facilitar a leitura e não possui significado sintático.

Após o identificador da extensão, é possível adicionar o número da versão implementada, e caso seja uma versão que utiliza um ponto como separador, é utilizado um ‘p’ no lugar, por exemplo, RV32I2p1 para a versão 2.1 do conjunto RV32I.

O núcleo desenvolvido neste trabalho implementa o conjunto RV32I com as extensões C e Zicsr e por essa razão pode ser identificado como um processador RV32ICZicsr.

Mnemônico	Argumentos	Definição
C.LWSP	$(rd, uimm^{'8})$	CI(2, 2)$(rd, uimm$[4:2|7:6]$, uimm$[5:5]$), $rd \neq 0$
C.SWSP	$(rs2, uimm^{'8})$	CSS(2, 6)$(rs2, uimm$[5:2|7:6]$)$
C.LW	$(rd', rs1', uimm^{'8})$	CL(0, 2)$(rd', rs1', uimm$[2:2|6:6]$, uimm$[5:3]$)$
C.SW	$(rs1', rs2', uimm^{'8})$	CS(0, 2)$(rs1', rs2', uimm$[2:2|6:6]$, uimm$[5:3]$)$
C.J	$(imm^{'12})$	CJ(1, 5)$(imm$[11:11|4:4|9:8|10:10|6:7|3:1|5:5]$)$
C.JAL	$(imm^{'12})$	CJ(1, 1)$(imm$[11:11|4:4|9:8|10:10|6:7|3:1|5:5]$)$
C.JR	$(rs1)$	CI(2, 4)$(rs1, 0, 0)$, $rs1 \neq 0$
C.JALR	$(rs1)$	CI(2, 4)$(rs1, 0, 1)$, $rs1 \neq 0$
C.BEQZ	$(rs1', imm^{'9})$	CB(1, 6)$(rs1, imm$[7:6|2:1|5:5]$, imm$[8:8|4:3]$)$
C.BNEZ	$(rs1', imm^{'9})$	CB(1, 7)$(rs1, imm$[7:6|2:1|5:5]$, imm$[8:8|4:3]$)$
C.LI	$(rd, imm^{'6})$	CI(1, 2)$(rd, imm$[4:0]$, imm$[5:5]$)$, rd \neq 0$
C.LUI	$(rd, nzimm^{'18})$	CI(1, 3)$(rd, imm$[16:12]$, imm$[17:17]$)$, rd \neq 2, rd \neq 0$
C.ADDI	$(rs1/rd, nzimm^{'6})$	CI(1, 0)$(rs1/rd, nzimm$[4:0]$, nzimm$[5:5]$)$, $rs1/rd \neq 0$
C.ADDI16SP	$(nzimm^{'10})$	CI(1, 3)$(2, imm$[16:12]$, imm$[17:17]$)$, $nzimm \neq 0$
C.ADDI4SPN	$(rd', nzuimm^{'10})$	CIW(0, 0)$(rd', nzuimm$[5:4|9:6|2:3]$)$, $nzuimm \neq 0$
C.SLLI	$(rs1/rd, nzimm^{'6})$	CI(2, 0)$(rs1/rd, nzimm$[4:0]$, nzimm$[5:5]$)$, $rs1/rd \neq 0$
C.SRLI	$(rs1'/rd', uimm^{'6})$	CA(1, $uimm$[4:2], $4^{'3}$ + $uimm[5]^{'1}$ + $0^{'2}$) $(rs1'/rd', uimm$[1:0]$)$, $\|uimm[5]\| = 0$
C.SRAI	$(rs1'/rd', uimm^{'6})$	CA(1, $uimm$[4:2], $4^{'3}$ + $uimm[5]^{'1}$ + $1^{'2}$) $(rs1'/rd', uimm$[1:0]$)$, $\|uimm[5]\| = 0$
C.ANDI	$(rs1'/rd', imm^{'6})$	CA(1, imm[4:2], $4^{'3}$ + $imm[5]^{'1}$ + $2^{'2}$) $(rs1'/rd', imm$[1:0]$)$
C.MV	$(rd, rs2)$	CI(2, 4)$(rd, rs2, 0)$, $rs2 \neq 0, rd \neq 0$
C.ADD	$(rs1, rs2)$	CR(2, 9)$(rs1, rs2)$, $rs1 \neq 0, rs2 \neq 0$
C.AND	$(rs1'/rd', rs2')$	CA(1, 3, 35)$(rs1'/rd', rs2')$
C.OR	$(rs1'/rd', rs2')$	CA(1, 2, 35)$(rs1'/rd', rs2')$
C.XOR	$(rs1'/rd', rs2')$	CA(1, 1, 35)$(rs1'/rd', rs2')$
C.SUB	$(rs1'/rd', rs2')$	CA(1, 0, 35)$(rs1'/rd', rs2')$
C.NOP	()	CI(1, 0)(0, 0, 0)
C.EBREAK	()	CR(2, 9)(0, 0)

Tabela 2.11: *Tabela de instruções comprimidas*

2.3 *Verilog*

É utilizada uma linguagem de descrição de *hardware* para descrever o comportamento do circuito e, com base na descrição, sintetizar uma configuração (ou *netlist*) para uso em FPGAs ou fabricação de *chips*.

Foi inventada em 1984 por Prabhu Goel, Phil Moorby, Chi-Lai Huang e Douglas Warmke (Times, 2005) enquanto trabalhavam na *Gateway Design Automation* para uso no simulador *Verilog-XL*. A *Verilog* é hoje um padrão de indústria, e aceita tanto em simuladores de código fechado quanto aberto e é compatível com a maioria dos sintetizadores de lógica resistor-transistor (RTL do inglês *resistor-transistor logic*) utilizados para a geração de *netlists*.

De modo simplificado ela permite a representação de circuitos através de:

- *Wires* que representam sinais lógicos cujo valor é derivado de outros *wires* e *regs*.
- *Reg* que representam *bits* capazes de reter seu valor até a próxima atribuição de valor.
- *Modules* (ou módulos) que representam uma coleção de *wires*, *regs*, outros *modules* previamente definidos e relações entre eles.

Wires podem ser combinados em vetores similares a vetores de *bits*. Através de declarações `assign` é possível definir o valor de *wires* com base em uma expressão de outros *wires* e *regs* com operações similares as descritas na Seção 2.2.5.

Regs também podem ser combinados em vetores similares a vetores de *bits*. Através de declarações `always` é possível definir a atribuição de valores condicionados a um evento. Declarações `always` utilizam uma sintaxe similar à sequência de declarações dentro do corpo de uma função na linguagem *C*.

Modules são compostos por um conjunto de *wires*, *regs*, declarações `assign`, declarações `always` e instância de *modules* internos. Além disso, eles possuem uma lista de *wires* e *regs*, que são usados como sinais de entrada e saída (*regs* só podem ser utilizados como saída) para interface com outros módulos.

2.3.1 Exemplos de programas *Verilog*

Contador

O Programa 2.1 apresenta um *module* de contador, onde o *reg* `contador` é a saída e os *wires* `reset` e `clock` são as entradas do módulo.

O trecho `@(posedge clock)` após a palavra-chave `always` determina que sempre que o sinal do `clock` for de 0 para 1, a expressão `contador <= reset ? 0 : contador + 1` é processada. `<=` é um operador de atribuição assíncrona e `reset ? 0 : counter + 1` é similar ao operador ternário da linguagem *C*, determinando que o `contador` passará a valer 0 caso o sinal `reset` esteja elevado (valendo 1) ou será incrementado caso contrário.

Programa 2.1 Exemplo de um contador em *Verilog*

```
1   module Contador(
2      contador,
3      reset,
4      clock
5   );
6
7   output reg [31:0]contador;
8   input reset;
9   input clock;
10
11  always @(posedge clock) contador <= reset ? 0 : contador + 1;
12
13  endmodule
```

Multiplexador

Programa 2.2 Exemplo de um multiplexador em *Verilog*

```
1   module Multiplexador(
2      saida,
3      entradas,
4      seletor,
5   );
6
7   output saida;
8   input [3:0]entradas;
9   input [1:0]seletor;
10
11  assign saida = entradas[seletor];
12
13  endmodule
```

O Programa 2.2 apresenta um *module* de multiplexador, onde o *wire* `saida` é a saída e os *wires* `entradas` e `seletor` são as entradas do módulo.

A expressão `saida = entradas[seletor]` após a palavra-chave `assign` determina que a saída sempre irá apresentar o valor do *wire* de `entradas` na posição determinada pelo valor do vetor de *wires* `seletor`.

Relógio

O Programa 2.3 apresenta um *module* de relógio, onde os *wires* `segundo` e `minutos` são as saída e os *wires* `clock` e `reset` são as entradas do módulo.

O *module* `Contador` é instanciado em `minutos` e em `segundos`. A saída `segundo` é igual a saída `segundos.contator` e a saída `minuto` é igual a saída `minutos.contator`.

Para atingir o comportamento de um relógio, o contador de segundos é reiniciado caso ele seja incrementado enquanto vale 59. Isso é feito através do *wire* `segundos_reset` que vale 1 ou quando `reset` vale 1 ou quando `segundo` vale 59. Além disso, o contador dos

Programa 2.3 Exemplo de um relógio em *Verilog*

```
1  module Contador(
2     contador,
3     reset,
4     clock
5  );
6
7  output reg [31:0]contador;
8  input reset;
9  input clock;
10
11 always @(posedge clock) contador <= reset ? 0 : contador + 1;
12
13 endmodule
14
15 module Relogio(
16    segundo,
17    minuto,
18    reset,
19    clock
20 );
21
22 output [31:0]segundo;
23 output [31:0]minuto;
24 input reset;
25 input clock;
26
27 wire minutos_clock;
28 wire segundos_reset;
29
30 Contador segundos(
31    .contador(segundo),
32    .reset(segundos_reset),
33    .clock(clock)
34 );
35
36 Contador minutos(
37    .contador(minuto),
38    .reset(reset),
39    .clock(minutos_clock)
40 );
41
42 assign segundos_reset = reset | (segundo == 59);
43 assign minutos_clock = reset ? clock : segundo != 59;
44
45 endmodule
```

minutos só deve incrementar quando segundos for incrementado de 59 para 0, isso é feito através do *wire* `minutos_clock` que funciona como um sinal de *clock* que só sobe quando os segundos forem de 59 para 0 quando o `reset` vale 0 e reflete o `clock` quando o `reset` vale 1.

2.3.2 Simulação de módulos em *Verilog*

A forma usual de simular módulos *Verilog* é através da escrita de um módulo que não possui nem sinais de entrada nem sinais de saída e instancia os módulos a serem simulados. Por meio de funcionalidades não sintetizáveis da linguagem é possível simular estímulos de sinais a serem enviados para os módulos instanciados e avaliar se o comportamento é o esperado.

Porém, a escrita desses módulos pode acabar sendo muito rígida ou contra produtiva devido a linguagem *Verilog* ser primariamente uma linguagem de descrição de *hardware* e não uma linguagem de programação de uso geral. Uma solução utilizada para amenizar as limitações da linguagem é o uso de extensões que permitem que o simulador carregue bibliotecas dinâmicas e então disponibilize funções definidas nelas para que sejam chamadas pelo código *Verilog* como em Dawson *et al.*, 1996.

Uma outra solução é a utilizada pelo programa *Verilator* criado em 1994 por Paul Wasson enquanto trabalhava na *Digital Equipment Corporation* (Snyder, 2021). Ao invés de utilizar um simulador para simular o código *Verilog*, o *Verilator* converte o código em uma classe *C++* que pode ser compilada e ligada a uma biblioteca fornecida pelo programa para simular o comportamento do circuito.

O fato do módulo ser transformado em uma classe *C++* permite que a simulação seja escrita em qualquer linguagem de programação que apresente meios de chamar código *C++* e simplifica a integração com sistemas de integração contínua e testes tradicionais.

2.4 Objective-C

Objetive-C é uma linguagem de programação de uso geral que adiciona à linguagem *C* um sistema de objetos e mensagens similar ao da linguagem *Smalltalk*. Criada por Brad Cox e Tom Love, inicialmente como um pré processador para a *C* em 1983 (Cox, 1983), ela evoluiu para uma linguagem própria e ganhou popularidade junto ao sucesso do *iPhone*, por ter sido a principal linguagem de desenvolvimento.

Similar à *Smalltalk*, o sistema de objetos é implementado de forma dinâmica, permitindo que classes sejam definidas e modificadas em tempo de execução. Assim, a linguagem depende do ambiente de execução para ter seu comportamento determinado. Dentre os principais ambientes de execução disponíveis existem o *Objective-C runtime* de código fechado da *Apple* e os projetos de código aberto *GNUStep*, *Mulle-Objc* e *ObjFW*.

A interoperabilidade com código em *C* e a simplicidade do modelo de orientação a objetos implementado faz com que a linguagem seja de fácil aprendizado para programadores que já conheçam *C* e buscam utilizar funcionalidades de linguagens orientadas a objetos.

2.4.1 Objetos e Classes

Objetos são representados através de ponteiros em *C* e podem receber mensagens através de uma sintaxe especial que utiliza colchetes. As mensagens são compostas pelo objeto que recebe a mensagem, separadores terminados com o caractere `:` caso precedam um argumento e os argumentos da mensagem.

Considerando `listaDeCompra` um objeto, a expressão `[listaDeCompra adicionaProdutoComNome:@"Laranjas"unidade:@"Dúzia"quantidade:2.0]` envia para o objeto a mensagem `adicionaProdutoComNome:unidade:quantidade:` com os argumentos `@"Laranjas"`, `@"Dúzia"` e `2.0`, em que `@"..."` representa um objeto de cadeia de caracteres.

As classes podem ser definidas através de uma sintaxe especial como ilustra o Programa 2.4. Entre `@interface` e `@end` são declarados os seletores (métodos) da classe e suas propriedades. Esse trecho costuma ser inserido em um arquivo de cabeçalho para que a classe seja referenciada por outras classes. Entre `@implementation` e `@end` são implementados os seletores (métodos) da classe e esse trecho costuma ser inserido em um arquivo de implementação com a extensão `.m`. Seletores da classe são prefixados por um `+` na declaração e na implementação e seletores de uma instância da classe são prefixados por um `-`.

2.4.2 Protocolos

Protocolo é um mecanismo de *Objective-C* que funciona de forma similar a interfaces em *Java*. Ele permite definir um conjunto de seletores e avaliar se uma classe implementa essas mensagens ou não. A linguagem provê uma sintaxe especial para declarar que uma classe expressa em sua interface conformidade com um dado protocolo.

Programa 2.4 Exemplo de uma classe em *Objective-C*

```
1   // Declaração de uma Classe que extende a ClasseBase
2   @interface ClasseDeExemplo: ClasseBase
3
4   @property TipoDaPropriedade nomeDaPropriedade;
5
6   + (TipoDevolvido)mensagemParaClasseComArgumento:(TipoDoArgumento)
        nomeDoArgumento;
7   + (TipoDevolvido)mensagemParaClasseSemArgumento;
8
9   - (void)mensagemParaInstanciaSemArgumentoNemValorDevolvido;
10
11  @end
12
13  // Implementação de uma Classe
14  @implementation ClasseDeExemplo
15
16  + (TipoDevolvido)mensagemParaClasseComArgumento:(TipoDoArgumento)
        nomeDoArgumento {
17    // Implementação da mensagem como uma função C.
18    // self neste contexto referencia a classe.
19    [self mensagemParaClasseSemArgumento];
20    // ...
21  }
22
23  + (TipoDevolvido)mensagemParaClasseSemArgumento {
24    //...
25  }
26
27  - (void)mensagemParaInstanciaSemArgumento {
28    // self neste contexto referencia a instância da classe.
29
30    // exemplo de leitura do valor de uma propriedade:
31    [self nomeDaPropriedade]
32
33    // exemplo de escrita do valor de uma propriedade:
34    [self setNomeDaPropriedade: novoValor]
35  }
36
37  @end
```

O uso de protocolos é importante pois permite a adição de comportamentos por composição ao invés de herança. O Programa 2.5 ilustra a declaração de um protocolo e sua conformidade por duas classes. Mesmo que ambas as declarações das classes não listem o seletor `- (double)area`, a conformidade com o protocolo `seletores` implica na existência de uma implementação para o seletor.

Programa 2.5 Exemplo do uso de protocolos em *Objective-C*

```
1  // Declaração de um protocolo que extende o protocolo NSObject
2  @protocol FiguraComArea <NSObject>
3
4  - (double)area;
5
6  @end
7
8  // Declaração de uma classe que atende aos requisitos do protocolo
       FiguraComArea
9  @interface Quadrado: NSObject <FiguraComArea>
10
11 @property double largura;
12
13 @end
14
15 // Declaração de outra classe que atende aos requisitos do protocolo
       FiguraComArea
16 @interface Circulo: NSObject <FiguraComArea>
17
18 @property double raio;
19
20 @end
```

2.4.3 Blocos

Block ou bloco é uma extensão desenvolvida pela *Apple* para as linguagens *C* e *Objetive-C* (Apple, 2014). Ela permite a declaração de *closures* que são capazes de capturar valores do contexto de criação e posteriormente serem chamados como funções de *C*. O Programa 2.6 ilustra a declaração do tipo de um bloco, sua definição e utilização através de um programa que cria blocos que devolvem um múltiplo do valor passado como argumento.

Programa 2.6 Exemplo do uso de blocos em *Objective-C*

```
#import <Foundation/Foundation.h>

// Definição do tipo de um bloco que recebe um inteiro e devolve um inteiro.
typedef int (^Multiplicador)(int);

Multiplicador criaMultiplicador(int multiplicador) {
  // Criação de um bloco que captura o valor do multiplicador
    Multiplicador bloco = ^(int valor) {
       return valor * multiplicador;
    };
    return bloco;
}

int main() {
   @autoreleasepool {
       Multiplicador dobro = criaMultiplicador(2);
       Multiplicador triplo = criaMultiplicador(3);
       printf("%d * 2 = %d\n", 5, dobro(5));
    // 5 * 2 = 10
       printf("%d * 3 = %d\n", 8, triplo(8));
    // 8 * 3 = 24
       printf("%d * 6 = %d\n", 7, triplo(dobro(7)));
    // 7 * 6 = 42
    }
    return 0;
}
```

2.5 Matriz de porta programáveis

Matriz de porta programáveis (FPGA) é um dispositivo constituído por blocos de lógica programável (CLB, do inglês *configurable logic block*), interconectores e blocos de entrada e saída (IOB, do inglês *input output block*) que são configurados a fim de implementar um dado circuito lógico.

A primeira FPGA, a *XC2064*, foi introduzida em 1984 pela *Xilinx* (Trimberger, 2018) contendo 64 CBLs *XCSP*. A *7 series*, uma família de FPGAs lançada em 2010 pela *Xilinx* (Xilinx, 2020) conta com modelos com até 305 mil CBLs (Xilinx, 2016), sendo as CLBs da *7 series* capazes de representar circuitos mais complexos que as CLBs da *XC2064*.

O alto custo inicial para a fabricação de um *chip* em relação à aquisição de FPGAs e o aumento dos circuitos que podem ser implementados nelas permitem que elas sejam usadas tanto no processo de desenvolvimento de um circuito integrado (CI) quanto no produto final, dependendo da escala do projeto, através da adição de alguma memória que guarda a configuração desejada para ser carregada ao ligar a FPGA.

2.5.1 Blocos de lógica programável

Um bloco de lógica programável de uma FPGA pode ser tão simples quanto um transistor ou tão complexo quanto um microprocessador, sendo capaz de implementar diferentes

circuitos combinacionais e sequenciais (Rose *et al.*, 1993).

Uma possibilidade de implementação de um CLB é o uso de uma tabela de pesquisa ou *lookup table* (LUT), combinadas com um *flip-flop* e um multiplexador. A tabela é configurada com valores para representar um circuito combinacional e os seletores da tabela são ligados aos sinais de entradas do CLB. A saída da tabela é ligada no *flip-flop* e tanto a saída da tabela quanto a saída do *flip-flop* são ligadas no multiplexador, cuja saída é ligada na saída da CLB. Um bit de configuração define o valor de saída do multiplexador e a entrada para alterar o valor do *flip-flop* é ligada aos sinais de entrada da CLB.

2.5.2 Interconectores

Interconectores são fios que ligam as diversas entradas e saídas das CLBs e IOBs entre si através de uma malha com roteamento configurável. Uma série de pontos de conexão onde fios se sobrepõem utilizam um *bit* de configuração para determinar se os fios devem ser conectados ou não.

Eles representam a maior parte da área de uma FPGA devido a possibilidade de conexões ser exponencial em relação ao número de sinais de CLBs e IOBs. Desse modo, as FPGAs implementam diversas estratégias para permitir o uso eficiente do espaço do *chip* e ainda permitir que blocos distantes sejam conectados entre si.

2.5.3 Blocos de entrada e saída

Blocos de entrada e saída são responsáveis por garantir que sinais externos sejam refletidos internamente dentro dos parâmetros de configuração da FPGA, e que os sinais internos sejam propagados para fora da FPGA com parâmetros de tensão e correntes desejados.

Capítulo 3

Projeto do processador e sistema de testes

Este capítulo inicia descrevendo o ambiente de execução do processador, e se estende em seu desenvolvimento detalhando a microarquitetura do computador, com a descrição das unidades que a compõe, e então, é apresentada a estrutura do projeto, o sistema de teste, e é encerrado com detalhes sobre o processo de sintetização.

3.1 EEI

Para o projeto, foi escolhido desenvolver um processador com um *hart* físico RV32ICZicsr e uma interface de comunicação serial para processamento de informações. O ambiente de execução suporta acessos não alinhados na memória e disponibiliza os níveis de privilégio de máquina e usuário.

O sistema é composto por um *hart* com extremidade *little-endian*. A memória possui quatro regiões identificadas pelos símbolos sA, sB, sC e sD. Considerando $mem^{8'2^{\mathrm{XLEN}}}$, a Tabela 3.1 descreve o intervalo de memória de cada região e sua funcionalidade. Intervalos da memória não cobertos por essas regiões sempre possuem o valor de 0 e o valor do *pc* não pode apontar para eles. A memória da região sB é utilizada para mapear a interface serial, o temporizador de máquina, o sistema de bancos de memória e os pinos digitais de entrada e saída.

Interface Serial

A interface serial é utilizada para comunicação assíncrona com dispositivos externos. Sua comunicação é feita através do envio e recebimento de *bytes* com uma velocidade de 9600 Baud utilizando o protocolo *universal asynchronous receiver-transmitter* (UART) sem *bits* de paridade e com 1 *bit* de parada. A comunicação utiliza filas de 8 bytes para entrada e saída, permitindo que a comunicação seja processada em blocos de 8 bytes. A interface é mapeada em sB[3:0], em que:

- sB[0] vale 1 caso existam *bytes* a serem lidos, 3 caso a fila de leitura esteja cheia e 0

Região	Intervalo	Descrição
sA	mem[h27F:h80]	Região apenas de leitura utilizada para armazenar o inicializador do sistema. O *pc* só pode apontar para essa região de memória caso ele esteja no nível de máquina.
sB	mem[hF0F:hF00]	Região utilizada para mapeamento de dispositivos de entrada e saída. O *pc* não pode apontar para essa região da memória e ela só pode ser lida ou escrita pelo *hart* caso ele esteja no nível de máquina.
sC	mem[h203FF:h2000]	Região de memória com possibilidade de leitura e escrita caso o *hart* esteja no nível de máquina. O *pc* só pode apontar para essa região de memória caso ele esteja no nível de máquina.
sD	mem[h803FF:h8000]	Região de memória com possibilidade de leitura e escrita independente do nível do *hart*. Destinada para armazenamento dos programas que executam sem privilégios.

Tabela 3.1: *Tabela de regiões da memória disponibilizadas pelo* EEI

caso não tenham bytes disponíveis para a leitura.

- sB[1] vale 1 a fila de envio tenha espaço, 3 caso a fila de vazio esteja vazia e 0 caso ela esteja cheia.
- sB[2], quando lido, devolver o valor da cabeça da fila de leitura e remove o valor da fila.
- sB[3], quando escrito, adiciona o valor na fila de envio.

A interface serial aceita apenas leitura e escrita através das instruções LB e SB.

Temporizador

O intervalo sb[11:4] é utilizado para armazenar o temporizador de máquina. Enquanto o valor dele for menor ou igual ao do contador de máquina, será emitida uma interrupção de temporizador de máquina.

Sistema de bancos

A memória da região sD utiliza um sistema de bancos em que trechos de memória de 512 *bytes* constituem um banco de memória. sB[14], que só pode ser lido, guarda a quantidade de bancos disponíveis. sB[12] indica o banco que é mapeado em sD[h1FF:0] e sB[13] indica o banco que é mapeado em sD[h3FF:h200]. Os bancos são indexados a partir de zero, assim os valores de sB[12] e sB[13] devem ser sempre menores que sB[14] e a tentativa de escrita de um valor inválido é ignorada. Além disso, sB[12] e sB[13] podem

apontar para o mesmo banco. O sistema de bancos aceita apenas leituras e escrita através das instruções LB e SB.

Pinos de entrada e saída

O byte sB[15] é utilizado para mapear pinos de entrada e saída, em que sB[15][7:4] é mapeado em pinos de saída e sB[15][3:0] é mapeado em pinos de entrada. O funcionamento desses pinos é descrito na Seção 3.5.

3.1.1 Inicialização

Ao iniciar, o *hart* pula para a posição h280 e começa a execução do programa inicializador em modo de máquina, com interrupções desativadas. O valor do temporizador de máquina é alterado pra hFFFFFFFFFFFFFFFF e sB[12] e sB[13] valem 0.

3.2 Design do processador

Para o desenvolvimento do processador foi escolhido criar uma microarquitetura *in-order multycycle*, como a descrita em S. Harris e D. Harris, 2021. Uma microarquitetura *in-order* implica que o processador não realiza reordenação de instruções e *multicycle* implica que a execução da instrução é dividida em estágios, permitindo que instruções diferentes levem tempos diferentes para serem executadas.

A Figura 3.1 apresenta o diagrama de estados implementado pelo processador. Quando inicializado, o processador entra no estado de *Inicialização*, em que seu estado interno é ajustado de acordo com a EEI definida. Após isso, ele passa para o estado de *Aquisição*, em que a instrução a ser executada é carregada da memória. Caso a instrução seja válida o *hart* passa para o estado de *Execução*, em que são realizadas as alterações necessárias do estado interno. Nos casos em que a instrução exige a leitura ou escrita de algum valor da memória, o estado é alterado para o estado de *Leitura/Escrita*, que realiza a operação, e então o processador volta para o estado de *Aquisição* para carregar a próxima instrução. Quando o acesso à memória não é necessário, o processador passa do estado de *Execução* direto para o de *Aquisição*. Caso seja necessário levantar alguma exceção ou interrupção, o processador passa para o estado de *Exceção*, que ajusta os valores dos CSRs de acordo com a especificação, e então volta para o processo de *Aquisição*.

Para implementar essa máquina de estados, o processador foi dividido em unidades de controle, decodificação, processamento aritmético, arquivo de registro e interface de memória.

3.2.1 Unidade de controle

A unidade de controle é responsável pelo processamento das instruções, alterando conforme a instrução os sinais de comandos para as outras unidades e, quando necessário, levanta exceções ou interrupções. Ela mantém o valor do *pc* e CSRs, bem como organiza as operações que envolvem acesso à memória. A unidade mantém um registrador interno de estado que reflete o estado do processador na máquina de estados, e com base nesse

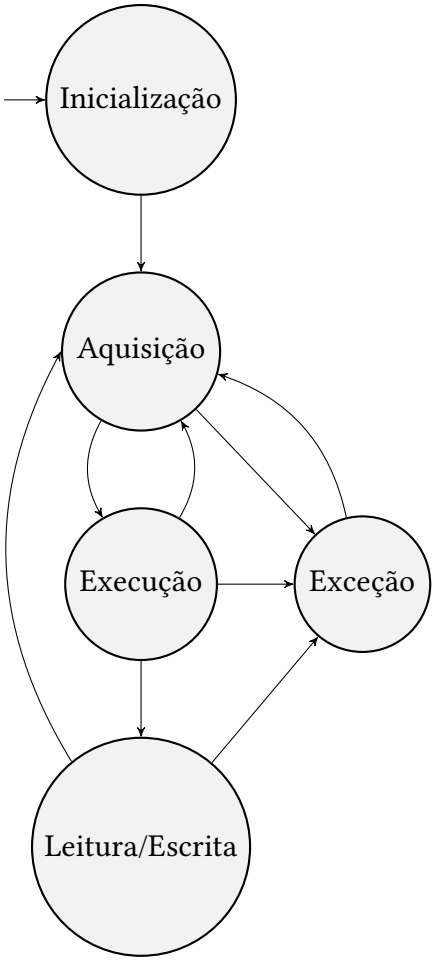

Figura 3.1: *Máquina de estados do processador*

registrador e em outros sinais de controle, a unidade atualiza seu estado durante a subida do *clock*. Ela também contém as outras unidades e realiza o roteamento necessário de sinais entre elas.

3.2.2 Unidade de decodificação

A unidade de decodificação é contida dentro da unidade de controle e ela é responsável por decodificar a instrução nas operações internas necessárias. Quando a instrução é obtida da memória, no estado de *Aquisição*, ela é passada para a unidade de decodificação, que, no primeiro estágio, determina se é uma instrução comprimida ou uma instrução de 32 *bits*. Caso seja uma instrução comprimida, é levantado um sinal para indicar que a instrução executada é uma instrução comprimida, garantido que o valor do próximo *pc* seja processado de acordo, e a instrução é mapeada em uma instrução de 32 *bits* para ser decodificada. Caso contrário, a instrução de 32 *bits* é decodificada diretamente.

Além de identificar se a operação executada é comprimida ou não, a unidade de decodificação determina a operação a ser realizada pela unidade de processamento aritmético e quais devem ser os valores utilizados como argumentos. Dentre os possíveis argumentos passados para a unidade de processamento aritmético estão alguns dos registradores, valor

do pc, valor do imediato codificado na instrução e as constantes 2 e 4. Caso a instrução exija uma operação de acesso a memória, a unidade de decodificação emite um sinal para indicar a operação e os detalhes dela. Ao sair do estado de *Execução*, a unidade de controle verifica esse sinal para determinar se passa para o estado de *Leitura/Escrita*.

3.2.3 Unidade de processamento aritmético

A unidade de processamento aritmético implementa as operações descritas na Seção 2.2.5. Para simplificar a decodificação, a unidade recebe como entrada os dois valores de 32 *bits* e os campos $funct3$ e $funct7[5]$, que são mapeados de acordo com as operações descritas pelas instruções. Assim, para processar a operação ADD(2, 3, 1), basta passar para a unidade de processamento aritmético o valor dos registradores $x2$ e $x3$ e os valores dos campos $funct3$ e $funct7[5]$ da instrução.

3.2.4 Arquivo de registro

O arquivo de registro é a unidade que contém o valor dos trinta e dois registradores. Ela possui três seletores de entrada, em que os dois primeiros determinam os registradores cujos valores são emitidos nas duas saídas de 32 *bits*. O terceiro é acompanhando de um sinal de entrada de 32 *bits* e, quando o *clock* sobe e o seletor não possui o valor 0, o valor do registrador é alterado de acordo com o valor do sinal de entrada.

3.2.5 Interface de memória

A interface de memória é similar à interface de memória utilizada no projeto PicoRV32 (Wolf, 2021). Ela é composta por três sinais do controlador para a interface e três sinais da interface para o controlador. O controlador emite os sinais de comando a ser executado, endereço de memória e valor a ser escrito. A interface emite os sinais de disponibilidade, interrupção e valor lido. O sinal de disponibilidade é mantido alto sempre que a interface pode receber um novo comando, e mantido baixo enquanto ela o processa. Quando um comando de leitura é processado e o sinal de disponibilidade fica alto, o valor é refletido no sinal de valor lido. O sinal de interrupção sobe quando a região processada é considerada inválida, como a escrita em uma região que não seja coberta pelas seções sA, sB, sC e sD.

O sinal de comando emitido pelo controlador pode ser de: “sem comando”, leitura de 1, 2 ou 4 *bytes* e escrita de 1, 2 ou 4 *bytes*. Os sinais de endereço de memória e valor a ser escrito são capturados pela interface quando o *clock* sobe, o sinal de comando é diferente de “sem comando” e o sinal de disponibilidade está alto.

3.2.6 Unidade de comunicação serial

A unidade de comunicação serial implementa o protocolo UART sem *bits* de paridade e 1 *bit* de parada sendo executado na velocidade de 9600 Baud. Os valores a serem enviados e lidos são mantidos em filas de 8 *bytes* que permitem inserção e remoção simultânea.

As linhas de comunicação são mantidas com o sinal alto enquanto não estiverem transmitindo valores. Quando um *byte B* é transmitido, primeiramente é transmitido um

bit de valor 0, depois os valores dos *bits* de B da menor posição para a maior e por fim um *bit* de valor 1. A lógica inversa é utilizada para receber um *byte*, primeiro detectando o momento em que o sinal de recepção vai do estado alto para o baixo, aguardando meio período de transmissão de um *bit* para então capturar o valor do sinal de transmissão nos próximos oito períodos em um registrador de deslocamento. O valor do registrador é colocado na fila e aguarda-se mais um período referente ao *bit* de parada. Cada *bit* enviado ou recebido dura 1/9600 segundos, como para cada *byte* são enviados 2 *bits* a mais, sendo a taxa de transmissão máxima de 960 *bytes* por segundo.

3.3 Estrutura do projeto

O projeto foi estruturado em quatro partes: a primeira agrupa os módulos escritos em *Verilog* e interfaces escritas em *C++* e *C* para definirem a interface dos módulos, a segunda é um conjunto de códigos em *C* e *Objective-C* que permitem que o comportamento dos módulos seja simulado utilizando *Objective-C*, a terceira parte é um conjunto de classes de apoio para simulação e a última é o sistema de testes.

3.3.1 Módulos *Verilog*

Os módulos em *Verilog* se localizam em uma pasta comum com arquivos de *C++* com o mesmo nome. Considerando o módulo *Exemplo*, o arquivo *Exemplo*`.v` contém o código em *Verilog* e o arquivo *Exemplo*`.cpp` o código em *C++*. Através de um arquivo *Makefile* (FSF, 2020) para automatizar o processo, é utilizado o programa *Verilator* para gerar classes *C++* que refletem a descrição do módulo. Utilizando as classes geradas, o arquivo *Exemplo*`.c`, através de uma classe intermediária, define as seguintes funções:

- `UHRMake`*Exemplo* que devolve um ponteiro opaco para uma instância da classe intermediária, que permite a simulação do circuito utilizando as classes geradas pelo *Verilator*.
- `UHRDestroy`*Exemplo* que recebe um ponteiro opaco da instância para ser destruído.
- `UHRPoke`*Exemplo* que recebe o ponteiro opaco, um número que representa um sinal de entrada ou um *reg* interno do módulo e um valor para alterar o valor do sinal utilizando as classes geradas pelo *Verilator*.
- `UHRPeek`*Exemplo* que recebe o ponteiro opaco, um número que representa um sinal de entrada ou saída ou um *reg* ou um *wire* interno e devolve o valor do sinal.
- `UHREval`*Exemplo* que recebe o ponteiro e uma marca de tempo e aplica todas as alterações de valor dos sinais geradas com função UHRPoke*Exemplo* desde a última chamada de `UHREval`*Exemplo*. Essas alterações são aplicadas de forma simultânea na marca de tempo passada como argumento e é computado o novo estado dos sinais no circuito pelas classes geradas pelo *Verilator*.

Além do arquivo *Exemplo*`.cpp`, existe um arquivo `UHRModule`*Exemplo*`Interface.h` que contém o protótipo das funções implementadas em *Exemplo*`.cpp` e uma enumeração que relaciona os valores para identificar os sinais em símbolos relacionados aos nomes utilizados no módulo.

As classes geradas e o arquivo *Exemplo*.c são compilados e empacotados em uma biblioteca. Esse processo é realizado para cada módulo e, por fim, é gerada uma biblioteca com o código auxiliar do *Verilator* necessário para executar as classes geradas.

Por convenção, os valores são manipulados utilizando inteiros *unsigned* de 32 *bits*, assim, sinais com mais de 32 *bits* de largura são divididos em sinais secundários para acesso de cada parte do valor.

3.3.2 Interface de módulos

Com base na interface descrita na Seção 3.3.1, para cada módulo existe uma classe em *Objective-C* que se conforma ao protocolo definido no Programa 3.1. Devido à estrutura muito similar entre os módulos, existe uma classe abstrata que implementa a lógica para chamar as funções e reduzir o código duplicado entre as classes.

Programa 3.1 Protocolo `UHRModuleInterface`

```
1  @protocol UHRModuleInterface <NSObject>
2
3  /// @brief: Change the value of a signal
4  - (void)pokeSignal:(UHREnum)signal withValue:(UHRWord)value;
5
6  /// @brief: Return the value of a signal
7  - (UHRWord)peekSignal:(UHREnum)signal;
8
9  /// @brief: Evaluate the simulation of the model at a given time
10 - (void)evaluateStateAtTime:(UHRTimeUnit)time;
11
12 /// @brief: Get the signal of for the clock wire
13 - (UHREnum)clockSignal;
14
15 /// @brief: Get the name of the signals
16 - (NSDictionary *)signalNames;
17
18 @end
```

3.3.3 Classes de apoio

Além das classes para simulação, o projeto possui algumas classes de apoio para atender a necessidades pontuais.

Sistema de despacho

Para simulação de memória, ao invés de descrever um grande vetor de *bytes*, existe um módulo que utiliza um *buffer* de memória e processa operações de leitura e escrita através de chamadas de funções de código de máquina. Considerando a natureza dinâmica da linguagem *Objetive-C*, foi construído um sistema de despacho de mensagens para implementar o comportamento do módulo.

Utilizando a interface *DirectC interface* (DPI-C) (Krishna, 2005) que permite que módulos chamem funções em *C*, foi exposta a função `UHRModuleDispatch` que aceita

como argumento cinco *inteiros* e devolve um. O primeiro argumento é o identificador da fonte, o segundo, o pedido, e os últimos três são argumentos da mensagem. Para que a mensagem seja direcionada corretamente, o módulo possui um campo *reg*, conhecido, capaz de guardar o identificador.

O Programa 3.2 apresenta a interface da classe de despacho que implementa o sistema. O seletor `defaultDispatch` permite que apenas uma instância da classe seja utilizada, e através do seletor `registerHandler:withContex:forID:` é devolvido um identificador que, quando utilizado para enviar uma mensagem, a redirecionará para o *handler* registrado. Assim, a função `UHRModuleDispatch` redireciona a mensagem através da instância padrão da classe, utilizando o seletor `dispatchFromSource:request:arg0:arg1:arg2:`.

No caso do módulo de memória, o objeto da classe que implementa o *buffer* de memória e o mecanismo para processar mensagens de leitura e escrita, ao ser instanciado, gera um identificador. Esse identificador é escrito através do seletor `pokeSignal:withValue:` no campo que guarda o identificador no módulo no começo da simulação, e assim as mensagens enviadas pelo módulo durante a simulação são direcionadas para o objeto adequado.

Programa 3.2 Interface da classe de despacho de mensagens

```
1   typedef unsigned int (*UHRModuleDispatchHandler) (unsigned int source,
        unsigned int request, unsigned int arg0, unsigned int arg1, unsigned int
        arg2, void * context);
2
3   @interface UHRModuleDispatchManager : NSObject
4
5   + (instancetype)defaultDispatch;
6
7   - (unsigned int)dispatchFromSource:(unsigned int)aSource
8                              request:(unsigned int)aRequest
9                                 arg0:(unsigned int)arg0
10                                arg1:(unsigned int)arg1
11                                arg2:(unsigned int)arg2;
12
13  - (unsigned int)registerHandler:(UHRModuleDispatchHandler)aHandler
14                       withContex:(void *)context
15                            forID:(unsigned int)anId;
16
17  - (unsigned int)removeHandlerForID:(unsigned int)anId;
18
19  @end
```

Memória simulada

A classe responsável pela simulação da memória é capaz de simular latência de leitura e escrita. Ao instanciar um objeto da classe, ele aceita como parâmetro o tempo de resposta em ciclos de *clock* e quando o módulo de memória recebe um comando, antes de enviar a mensagem para executar o comando, é enviada uma mensagem para obter a latência que é atribuída a um contador do módulo que decrementa conforme os ciclos. No ciclo em

que o contador passa a valer 0 o módulo de memória envia a mensagem para processar o comando.

Mini montador

O mini montador é uma classe que possui seletores que devolvem quase todas as instruções de 32 *bits* do conjunto RV32I. A sintaxe é baseada no formato apresentado na Seção 2.2.5 com adaptações para as convenções da *Objective-C*. Assim ADDI(2, 3, 10) pode ser obtido com a expressão `[UHRRISCVMiniAssembler addiWithRD:2, rs1:3 imm:10]`. Seu principal uso é para tornar mais clara a escrita de testes que fazem uso de instruções.

3.4 Sistema de testes

Os testes são utilizados para validar se o comportamento de um módulo em uma situação específica reflete o comportamento esperado. Isso envolve a simulação do circuito com uma série de estímulos e verificações do valor de sinais em tempos específicos da simulação. A simulação gerada pelo *Verilator* utiliza precisão de ciclos e não temporal, assim o tempo da simulação é demarcado com base na subida e descida do *clock*, e não no tempo simulado transcorrido.

Para descrever os estímulos e verificações, é utilizada uma abstração de roteiro. Com base em um dicionário que o descreve, é gerado um objeto que executa as atividades descritas no modelo. A execução da simulação é efetuada por um objeto, chamado de *TestBench*, que avança a simulação e utiliza o objeto que executa o roteiro. Além do objeto que executa roteiros e o *TestBench*, é utilizado o arcabouço *XCTest* (Apple, 2021b) para executar e organizar os resultados dos testes.

3.4.1 Roteiro de testes

A simulação é baseada na alternância do valor do sinal do *clock* e cada ciclo do roteiro é composto pelo momento em que o *clock* vai de 0 para 1 (subida ou *rise*) e pelo momento em que ele vai de 1 para 0 (queda ou *fall*). Em um ciclo qualquer, o valor dos sinais pode ser alterado logo antes do momento da subida ou da descida e o valor dos sinais pode ser verificado após a subida ou descida do *clock*.

Desse modo, o roteiro é descrito por um dicionário cujas chaves são ciclos em que as operações devem ser executadas, e o valor é um dicionário que descreve as operações. Caso seja alterado o valor de algum sinal logo antes da subida do *clock*, o valor da chave representada pela sequência de caracteres `applyOnRise` é um vetor que contém o sinal no índice par e o valor a ser aplicado no índice ímpar. Caso a alteração ocorra logo antes da descida do *clock*, um vetor similar é armazenado na chave `applyOnFall`. Para as verificações após a subida e após a descida é utilizado o mesmo formato de vetor nas chaves `checkOnHigh` e `checkOnLow`.

Além das chaves `applyOnRise`, `applyOnFall`, `checkOnHigh` e `checkOnLow`, caso a chave `pass` exista, o roteiro é considerado concluído, e a chave `callback` pode armazenar

um bloco que recebe como argumento o módulo e o tempo da execução, para realizar qualquer computação arbitrária após a aplicação dos sinais e as verificações.

3.4.2 *Testbench*

O *TestBench* é inicializado com a instância do módulo a ser simulado e o executor do roteiro. Após a inicialização é possível executar a simulação até uma quantidade desejada de ciclos. Ela consiste em um laço que realiza as seguintes ações em ordem:

1. O *clock* do módulo é alterado para 1.
2. São aplicadas as alterações de subida caso existam para a marca de tempo.
3. É computado o novo estado do módulo através do seletor `evaluateStateAtTime:`.
4. Caso existam verificações após subida para o ciclo, elas são executadas.
5. O *clock* do módulo é alterado para 0.
6. São aplicadas as alterações de descida caso existam para a marca de tempo.
7. É computado o novo estado do módulo através do seletor `evaluateStateAtTime:`.
8. Caso existam verificações após descida para o ciclo, elas são executadas.
9. É chamado o *callback*, caso presente.
10. É verificado se o roteiro foi concluído e caso isso ocorra, o laço é parado.

Onde o tempo de simulação durante a subida é dado pelo número da iteração (começando do 1) vezes 10 e o da descida é dado pelo número da iteração vezes 15. O ciclo é dado pelo número da iteracão menos 1.

3.4.3 Execução de testes e integração contínua

Os testes foram feitos com o arcabouço *XCTest*. Ele utiliza o modelo baseado em testes de unidade onde classes que estendem a classe base `XCTestCase` são usadas para escrever os testes. Todos os seletores de instância dessas classes que não recebem argumentos e começam com o termo `test` são considerados testes. As classes devem definir implementações para os seletores de instância `setUp` e `tearDown` que são chamadas, respectivamente, antes e depois da execução de cada teste.

Além da classe de teste, são disponibilizadas funções que recebem uma condição e uma mensagem de erro como entrada, e caso a condição seja falsa, durante a execução do teste é registrado que a condição falhou, e com isso o teste também falha, porém, sua execução não é interrompida.

O *XCTest* disponibiliza um binário para rodar os testes. Ele carrega de forma dinâmica as bibliotecas com as classes e suas dependências e identifica em tempo de execução as classes de teste o os métodos a serem executados. Através do *Xcode* (Apple, 2021a) que possui integração com o *XCTest*, as bibliotecas são combinadas e é executado um plano de testes que compreende todas as classes.

O *Xcode* possui um modo de execução como servidor e ele foi utilizado para executar conjunto de testes conforme o repositório que armazena o código do repositório é atualizado. A Figura 3.2 apresenta a visão do painel de integrações (execuções) do conjunto de testes, onde é possível observar na seção superior a quantidade de testes que passaram na última integração, e, no gráfico inferior, o histórico dos testes executados em cada integração.

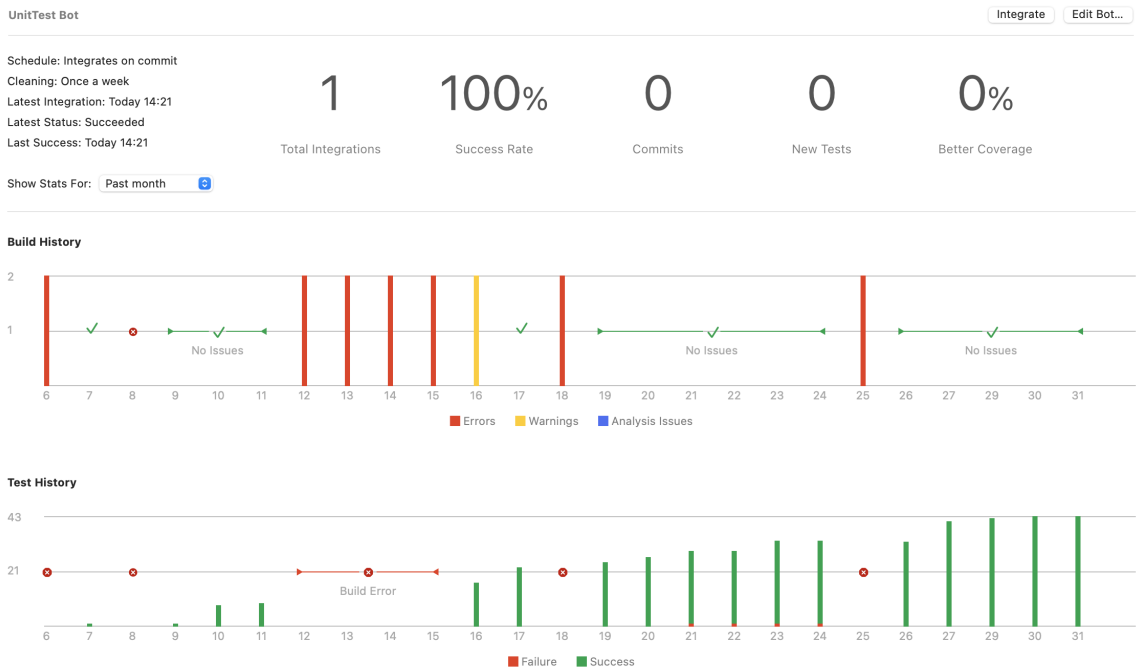

Figura 3.2: *Painel de integração do* Xcode

O *TestBench* e a classe de execução de roteiros foram desenvolvidas com a interface do *XCTest* em mente e as usam para apresentar mensagens de falha quando uma verificação falha. A Figura 3.3 apresenta um exemplo de mensagens de falha durante a execução de um teste. Na imagem, é possível observar que, após a subida do *clock* no ciclo oito, o sinal referente ao registrador $x01$ possuía valor 129 ao invés do esperado de 128, e a falha foi cascateada em outros erros onde o valor esperado era uma unidade menor que o valor lido.

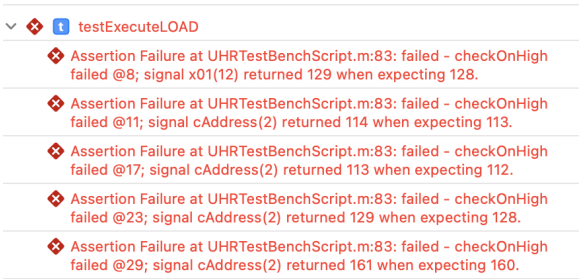

Figura 3.3: *Exemplo de mensagens de falha de verificação*

3.5 Sintetização

O processador foi sintetizado para uso em uma placa de desenvolvimento que utiliza a FPGA *XC7A100T-FGG676ABX1901* da *Xilinx* e possui o módulo *CP2102N* da *Silicon Labs* para passar a comunicação UART através de um cabo USB. A placa também provê um oscilador de cristal que gera um sinal de *clock* na frequência de 50 *Mhz*, dois botões e dois diodos emissores de luz.

Para programar a FPGA foi utilizado o *Vivado 2018*, uma suíte de design da *Xilinx* que possui os programas necessários para sintetizar a configuração a partir dos módulos e programar o dispositivo. Além do código em *Verilog*, foi necessário escrever um arquivo *Xilinx Design Constrain* (XDC) (Xilinx, 2018) que relaciona a entradas e saídas do módulo de nível mais alto com pinos da FPGA. Com base na folha de especificação da placa de desenvolvimento, foi relacionado quais pinos da *FPGA* representam os dois fios necessários para utilizar o módulo UART, o gerador de *clock*, os diodos emissores de luz e os botões. Com base na descrição dos módulos e o arquivo XDC, o *Vivado* gera um arquivo de configuração. Ao configurar a FPGA com esse arquivo através de uma interface JTAG, ela passa a funcionar como o processador descrito.

Em relação aos pinos de entrada e saída, o primeiro botão foi mapeado no *bit* sB[15][0] e os diodos foram mapeados nos *bits* sB[15][5] e sB[15][4]. Os outros pinos não foram mapeados em nenhum elemento da placa, e o segundo botão foi mapeado no sinal de *reset* para inicializar o processador. Devido a limitação de memória interna da FPGA, foram disponibilizado apenas dois bancos de memória para uso na região sD.

Capítulo 4

Conclusão

Desenvolver um processador RISC-V é uma tarefa extensa, que exige uma ampla gama de conhecimento para ser realizada. A princípio, a ISA ser aberta se mostra como fator convidativo, em contraste com conjuntos de instruções mais populares, que costumam ser proprietários, como ARM e x86. No entanto, a RISC-V é flexível para atender casos de uso bastante variados e para oferecer a tal característica, coloca no desenvolvedor a responsabilidade de tomar diversas decisões acerca do ambiente de execução, tais como: *endianness*, organização da memória e mapeamento de dispositivos de entrada e saída. A fim de agilizar o projeto, o desenvolvedor pode optar por utilizar os padrões definidos em outras implementações ou até mesmo iniciar sua própria implementação a partir de uma de código já existente.

Definido o conjunto de instruções oferecidos pelo processador, é preciso ponderar sobre a maneira com que as instruções serão executadas, os requisitos do projeto devem guiar essa escolha. Apesar de técnicas como *pipelining* oferecerem ganhos substanciais de desempenho, o projeto é penalizado com o aumento de complexidade, prolongando sua execução. Portanto, é uma boa prática ater-se à uma microarquitetura menos complexa, contanto que atenda aos requisitos impostos.

Com os detalhes do processador definidos, o projeto deve ser estruturado de modo adequado para o desenvolvimento e verificação dos módulos. Enquanto o uso de uma HDL costuma ser satisfatório para a descrição do *hardware*, a verificação é compreendida por etapas em que apenas o seu uso pode ser contra produtivo. Devido ao aumento exponencial do número de estados em relação ao aumento do número de células de memória no projeto, o tempo gasto na verificação acaba sendo muito maior que o tempo gasto na descrição dos componentes, e assim, a dedicação de um tempo maior para o desenvolvimento de um sistema de testes mais sofisticado é justificada com a redução no tempo para a escrita das verificações.

4.1 Próximos passos

Em relação ao processador, é possível adicionar mais extensões, como a extensão M, que acelera operações matemáticas com valores inteiros, e expandir o modelo da memória

para ter acesso a outros dispositivos de entrada e saída, ou módulos de memória DDR3, aumentando o espaço disponível para programas. No que se refere ao sistema de teste, é interessante explorar em iterações futuras a implementação de roteiros mais complexos para simplificar a escrita de testes que fazem uso de técnicas, como *fuzzing*, utilizada para cobrir um número maior de estados durante os testes.

Referências

[Apple 2014] Apple. *Working with Blocks*. [Online; Acessado em 21/12/2021]. 2014. URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithBlocks/WorkingwithBlocks.html (citado na pg. 29).

[Apple 2021a] Apple. *Xcode*. [Online; Acessado em 22/12/2021]. 2021. URL: https://developer.apple.com/xcode/ (citado na pg. 42).

[Apple 2021b] Apple. *XCTest*. [Online; Acessado em 21/12/2021]. 2021. URL: https://developer.apple.com/documentation/xctest (citado na pg. 41).

[Asanović e Patterson 2014] Krste Asanović e David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Rel. técn. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html (citado na pg. 1).

[Chen *et al.* 2020] Chen Chen *et al.* "Xuantie-910: a commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension : industrial product". Em: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pgs. 52–64. DOI: 10.1109/ISCA45697.2020.00016 (citado na pg. 1).

[RISC-V 2021a] RISC-V Software Colaboration. *RISC-V GNU Toolchain*. [Online; Acessado em 16/12/2021]. 2021. URL: https://github.com/riscv-collab/riscv-gnu-toolchain (citado na pg. 2).

[Cox 1983] Brad L. Cox. "The object oriented pre-compiler: programming smalltalk 80 methods in c language". Em: *SIGPLAN Not.* 18.1 (1983), pgs. 15–22. ISSN: 0362-1340. DOI: 10.1145/948093.948095. URL: https://doi.org/10.1145/948093.948095 (citado na pg. 27).

[Dawson *et al.* 1996] C. Dawson, S.K. Pattanam e D. Roberts. "The verilog procedural interface for the verilog hardware description language". Em: *Proceedings. IEEE International Verilog HDL Conference*. 1996. DOI: 10.1109/IVC.1996.496013 (citado na pg. 26).

[Debian 2021] Debian. *RISC-V*. [Online; Acessado em 16/12/2021]. 2021. URL: https://wiki.debian.org/RISC-V (citado na pg. 2).

[Elsabbagh *et al.* 2020] Fares Elsabbagh *et al.* *Vortex: OpenCL Compatible RISC-V GPGPU*. 2020. arXiv: 2002.12151 [cs.DC] (citado na pg. 1).

[Etnus 2003] Etnus. *MIPS Delay Slot Instructions*. [Online; Acessado em 16/12/2021]. 2003. URL: http://www.jaist.ac.jp/iscenter-/mpc/old-machines/altix3700/opt/toolworks/totalview.6.3.0-1/doc/html/ref_guide/MIPSDelaySlotInstructions.html (citado na pg. 1).

[Fisher 1983] Joseph A. Fisher. "Very long instruction word architectures and the eli-512". Em: *SIGARCH Comput. Archit. News* 11.3 (1983), pgs. 140–150. ISSN: 0163-5964. DOI: 10.1145/1067651.801649. URL: https://doi.org/10.1145/1067651.801649 (citado na pg. 7).

[FSF 2020] Free Software Foundation. *GNU Make Manual*. [Online; Acessado em 21/12/2021]. 2020. URL: https://www.gnu.org/software/make/manual/ (citado na pg. 38).

[S. Harris e D. Harris 2021] S.L. Harris e D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021. ISBN: 9780128200650. URL: https://books.google.com.br/books?id=SksiEAAAQBAJ (citado na pg. 35).

[Krishna 2005] Gopi Krishna. *System Verilog DPI Tutorial*. [Online; Acessado em 21/12/2021]. 2005. URL: http://www.testbench.in/DP_00_INDEX.html (citado na pg. 39).

[Pong 2017] Siu Ching Pong. *Go (Golang) GOOS and GOARCH*. [Online; Acessado em 16/12/2021]. 2017. URL: https : / / gist . github . com / asukakenji / f15ba7e588ac42795f421b48b8aede63 (citado na pg. 2).

[RISC-V 2021b] RISC-V. *RISC-V International – History*. [Online; Acessado em 16/12/2021]. 2021. URL: https://riscv.org/about/history/ (citado na pg. 1).

[RISC-V 2021c] RISC-V. *RISC-V International – Members*. [Online; Acessado em 15/12/2021]. 2021. URL: https://riscv.org/members/ (citado na pg. 1).

[Rose *et al.* 1993] Jonathan Rose, Abbas El Gamal e Alberto Sangiovanni-Vincentelli. "Architecture of field-programmable gate arrays". Em: *Proceedings of the IEEE* 81.7 (1993), pgs. 1013–1029 (citado na pg. 31).

[SiFive 2021] SiFive. *SiFive's LLVM*. [Online; Acessado em 16/12/2021]. 2021. URL: https://github.com/sifive/riscv-llvm (citado na pg. 2).

[Snyder 2021] Wilson Snyder. *Verilator Manual Version - Contributors and Origins*. [Online; Acessado em 20/12/2021]. 2021. URL: https://veripool.org/guide/latest/contributors.html (citado na pg. 26).

[Times 2005] EE Times. *Verilog's inventor nabs EDA's Kaufman award*. [Online; Acessado em 20/12/2021]. 2005. URL: https://www.eetimes.com/verilogs-inventor-nabs-edas-kaufman-award/ (citado na pg. 23).

[Trimberger 2018] Stephen M. Steve Trimberger. "Three ages of fpgas: a retrospective on the first thirty years of fpga technology: this paper reflects on how moore's law has driven the design of fpgas through three epochs: the age of invention, the age of expansion, and the age of accumulation". Em: *IEEE Solid-State Circuits Magazine* 10.2 (2018), pgs. 16–29. DOI: 10.1109/MSSC.2018.2822862 (citado na pg. 30).

[Waterman e Asanović 2019a] Andrew Waterman e Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 20191213*. Rel. técn. RISC-V International, 2019 (citado nas pgs. 6, 11, 16).

[Waterman e Asanović 2019b] Andrew Waterman e Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: User-Level ISA, Version 20190608-Priv-MSU-Ratified*. Rel. técn. RISC-V International, 2019 (citado nas pgs. 10, 18).

[Wolf 2021] Clifford Wolf. *PicoRV32 - A Size-Optimized RISC-V CPU*. [Online; Acessado em 21/12/2021]. 2021. URL: https://github.com/YosysHQ/picorv32 (citado na pg. 37).

[Xilinx 2016] Xilinx. *7 Series FPGAs Configurable Logic Block*. 2016 (citado na pg. 30).

[Xilinx 2018] Xilinx. *Vivado Design Suite User Guide*. 2018 (citado na pg. 44).

[Xilinx 2020] Xilinx. *7 Series FPGAs Data Sheet: Overview*. 2020 (citado na pg. 30).

[Yosys 2019] Yosys. *RISC-V*. [Online; Acessado em 16/12/2021]. 2019. URL: https://github.com/YosysHQ/picorv32 (citado na pg. 1).