



## TRABALHO DE GRADUAÇÃO

### RISC-V SiMPLÉ: PROJETO E DESENVOLVIMENTO DE PROCESSADORES RISC-V COM A ISA RV32IMF USANDO AS MICROARQUITETURAS UNICICLO, MULTICICLO E PIPELINE EM FPGA

Arthur de Matos Beggs

Brasília, Maio de 2021



**ENGENHARIA  
MECATRÔNICA**  
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA  
Faculdade de Tecnologia  
Curso de Graduação em Engenharia de Controle e Automação

## TRABALHO DE GRADUAÇÃO

### RISC-V SiMPLE: PROJETO E DESENVOLVIMENTO DE PROCESSADORES RISC-V COM A ISA RV32IMF USANDO AS MICROARQUITETURAS UNICICLO, MULTICICLO E PIPELINE EM FPGA

Arthur de Matos Beggs

*Relatório submetido como requisito parcial de obtenção  
de grau de Engenheiro de Controle e Automação*

Banca Examinadora

Prof. Dr. Marcus Vinicius Lamar, CIC/UnB \_\_\_\_\_  
*Orientador*

Prof. Dr. Ricardo Pezzuol Jacobi, CIC/UnB \_\_\_\_\_  
*Examinador Interno*

Prof. Dr. Marcelo Grandi Mandelli, CIC/UnB \_\_\_\_\_  
*Examinador Interno*

Brasília, Maio de 2021

## FICHA CATALOGRÁFICA

ARTHUR, DE MATOS BEGGS

RISC-V SiMPLE,

[Distrito Federal] 2021.

???, ???p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2021). Trabalho de Graduação – Universidade de Brasília. Faculdade de Tecnologia.

1. RISC-V

2. Verilog

3. FPGA

I. Mecatrônica/FT/UnB

II. Título (Série)

## REFERÊNCIA BIBLIOGRÁFICA

BEGGS, ARTHUR DE MATOS, (2021). RISC-V SiMPLE. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-nº???, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, ???p.

## CESSÃO DE DIREITOS

AUTOR: Arthur de Matos Beggs

TÍTULO DO TRABALHO DE GRADUAÇÃO: RISC-V SiMPLE.

GRAU: Engenheiro

ANO: 2021

É concedida à Universidade de Brasília permissão para reproduzir cópias deste Trabalho de Graduação e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte desse Trabalho de Graduação pode ser reproduzida sem autorização por escrito do autor.

---

Arthur de Matos Beggs

SHCGN 703 Bl G N° 120, Asa Norte

70730-707 Brasília – DF – Brasil.

---

## RESUMO

Desenvolvimento e documentação de uma plataforma de ensino de arquitetura de computadores em *Verilog* sintetizável em *FPGA*, com foco em um processador com arquitetura do conjunto de instruções *RISC-V* implementado em três microarquiteturas para ser utilizado como recurso de laboratório na disciplina de Organização e Arquitetura de Computadores da Universidade de Brasília. A plataforma funciona nas *FPGAs terasIC DE1-SoC* disponíveis no laboratório da Universidade, possui periféricos de depuração como *display* dos registradores do processador na saída de vídeo, além de outros periféricos como *drivers* de áudio e teclado para uma experiência mais completa de desenvolvimento, e permite que o processador seja substituído por implementações de diversas arquiteturas de *32 bits* com certa facilidade.

Palavras Chave: RISC-V, Verilog, FPGA

---

## ABSTRACT

Development and documentation of a computer architecture learning environment in Verilog synthesizable to FPGA, focusing in a computer processor using the RISC-V instruction set architecture implemented in three different microarchitectures. The project will be used as a lab resource on the Computer Architecture and Organization course at Universidade de Brasília. The platform works in FPGAs terasIC DE1-SoC available at the university lab, possess debugging peripherals such as an On Screen Display showing the contents of regfiles via video output, and other peripherals as audio and keyboard drivers delivering a full development experience. It allows with relative easiness the exchange of the core processor for other implementations of 32 bits architectures.

Keywords: RISC-V, Verilog, FPGA

# SUMÁRIO

<b>1</b>	<b>Introdução.....</b>	<b>1</b>
1.1	MOTIVAÇÃO .....	1
1.2	POR QUE RISC-V?.....	1
1.3	OBJETIVOS .....	2
1.4	ESTRUTURA DO TEXTO.....	2
<b>2</b>	<b>Revisão Teórica.....</b>	<b>3</b>
2.1	ARQUITETURA DE COMPUTADORES .....	3
2.1.1	ARQUITETURAS <i>RISC</i> e <i>CISC</i> .....	4
2.1.2	ARQUITETURA MIPS.....	5
2.1.3	ARQUITETURA ARM .....	5
2.1.4	ARQUITETURA X86.....	6
2.1.5	ARQUITETURA AMD64 .....	6
2.2	A ARQUITETURA RISC-V .....	6
2.2.1	EXTENSÃO DE REGISTRADORES DE CONTROLE E ESTADO .....	7
2.2.2	EXTENSÃO DE MULTIPLICAÇÃO E DIVISÃO.....	7
2.2.3	EXTENSÕES DE PONTO FLUTUANTE .....	8
2.2.4	EXTENSÕES A E ZIFENCEI.....	8
2.2.5	NOMENCLATURA DA IMPLEMENTAÇÃO .....	8
2.2.6	OUTRAS EXTENSÕES.....	8
2.2.7	ARQUITETURA PRIVILEGIADA .....	9
2.2.8	FORMATOS DE INSTRUÇÕES.....	9
2.2.9	FORMATOS DE IMEDIATOS .....	10
2.3	<i>IDE RARS</i> .....	11
2.4	MICROARQUITETURAS .....	12
2.4.1	UNICICLO .....	13
2.4.2	MULTICICLO.....	13
2.4.3	<i>PIPELINE</i> .....	13
2.5	REPRESENTAÇÃO DE <i>HARDWARE</i> .....	14
2.5.1	<i>VERILOG</i> .....	15
2.5.2	ANÁLISE E SÍNTSE.....	15
2.5.3	<i>FITTING</i> e <i>TIMING ANALYZER</i> .....	15
2.6	FIELD PROGRAMMABLE GATE ARRAYS.....	16

2.6.1	ARQUITETURA GENERALIZADA DE UMA FPGA .....	16
2.6.2	ARQUITETURA DA FPGA CYCLONE V SOC .....	17
2.6.2.1	ADAPTATIVE LOGIC MODULES .....	18
2.6.2.2	EMBEDDED MEMORY BLOCKS .....	19
2.7	ESTADO DA ARTE DOS PROCESSADORES RISC-V .....	19
2.8	OBSERVAÇÕES FINAIS DA REVISÃO TEÓRICA .....	20
<b>3</b>	<b>Sistema Proposto .....</b>	<b>21</b>
3.1	ORGANIZAÇÃO DO PROJETO .....	21
3.2	IMPLEMENTAÇÃO DOS <i>SOFT-CORES</i> .....	24
3.2.1	MICROARQUITETURA UNICICLO .....	24
3.2.2	MICROARQUITETURA MULTICICLO .....	26
3.2.3	MICROARQUITETURA <i>Pipeline</i> DE 5 ESTÁGIOS .....	28
3.3	CHAMADAS DE SISTEMA .....	30
3.4	UTILIZANDO O RARS .....	33
3.5	INTERFACE DE VÍDEO E DEPURAÇÃO .....	34
3.6	CONFIGURAÇÃO E SÍNTESE DO PROCESSADOR PELO QUARTUS .....	36
3.7	SIMULAÇÃO DO PROCESSADOR PELO QUARTUS E MODELSIM .....	37
3.8	SCRIPT <i>MAKE.SH</i> .....	39
3.9	USO DA FPGA DE1-SoC .....	39
3.10	OBSERVAÇÕES FINAIS DO SISTEMA PROPOSTO .....	40
<b>4</b>	<b>Resultados .....</b>	<b>41</b>
4.1	SÍNTESE DOS <i>SOFT-CORES</i> .....	41
4.2	FORMAS DE ONDA DAS SIMULAÇÕES .....	42
4.3	<i>BENCHMARKS</i> .....	44
4.4	OBSERVAÇÕES FINAIS DOS RESULTADOS .....	45
<b>5</b>	<b>Conclusões .....</b>	<b>46</b>
5.1	OBJETIVOS ALCANÇADOS .....	46
5.2	PERSPECTIVAS FUTURAS .....	47
5.3	PALAVRAS FINAIS .....	47
<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>		<b>48</b>
<b>Anexos .....</b>		<b>52</b>
<b>I Programas utilizados .....</b>		<b>53</b>
<b>II Diagramas das Microarquiteturas .....</b>		<b>54</b>

# LISTA DE FIGURAS

2.1	Abstração da arquitetura de um computador .....	4
2.2	Codificação de instruções de tamanho variável da arquitetura <i>RISC-V</i> .....	7
2.3	Formatos de Instruções da <i>ISA RISC-V</i> .....	9
2.4	Formatos de Instruções da <i>ISA MIPS32</i> .....	10
2.5	Formação do Imediato de tipo I .....	10
2.6	Formação do Imediato de tipo S.....	10
2.7	Formação do Imediato de tipo B .....	11
2.8	Formação do Imediato de tipo U .....	11
2.9	Formação do Imediato de tipo J .....	11
2.10	<i>IDE RARS</i> .....	12
2.11	<i>Pipeline</i> genérico com 4 estágios.....	14
2.12	Abstração da arquitetura de uma <i>FPGA</i> .....	17
2.13	Funcionamento da chave de interconexão .....	17
2.14	Arquitetura da <i>FPGA</i> Intel Cyclone V SoC .....	18
2.15	Diagrama de blocos de um ALM .....	18
3.1	Diagrama de blocos simplificado do sistema .....	24
3.2	Diagrama da implementação das <i>ISAs</i> RV32I e RV32IM na microarquitetura uniciclo	25
3.3	Diagrama da implementação da <i>ISA</i> RV32IMF na microarquitetura uniciclo .....	26
3.4	Diagrama da implementação das <i>ISAs</i> RV32I e RV32IM na microarquitetura multiciclo .....	27
3.5	Diagrama da implementação da <i>ISA</i> RV32IMF na microarquitetura multiciclo .....	28
3.6	Diagrama das <i>ISAs</i> RV32I e RV32IM na microarquitetura <i>pipeline</i> de 5 estágios .....	29
3.7	Diagrama da <i>ISA</i> RV32IMF na microarquitetura <i>pipeline</i> de 5 estágios .....	30
3.8	Exibição do <i>frame</i> de vídeo da <i>FPGA</i> .....	34
3.9	Exibição do <i>frame</i> de vídeo da <i>FPGA</i> .....	35
3.10	<i>Menu OSD</i> exibindo os valores dos registradores do processador .....	35
3.11	<i>Intel Quartus Lite v18.1</i> com a janela de configurações do projeto .....	36
3.12	Janela de configuração da simulação no <i>Quartus</i> .....	37
3.13	O <i>script</i> <i>NativeLink</i> invoca o <i>ModelSim</i> passando o <i>script</i> <i>.do</i> com as informações de como simular o sistema .....	37
3.14	Visualização das formas de onda no <i>GTKWave</i> .....	38
3.15	Placa de desenvolvimento <i>terasIC DE1-SoC</i> .....	40

4.1	Visualização das formas de onda <i>soft-core RV32IMF</i> uniciclo .....	43
4.2	Visualização das formas de onda <i>soft-core RV32IMF</i> multiciclo .....	43
4.3	Visualização das formas de onda <i>soft-core RV32IMF pipeline</i> .....	44
II.1	Diagrama do uniciclo <i>RV32I e RV32IM</i> .....	54
II.2	Diagrama do uniciclo <i>RV32IMF</i> .....	55
II.3	Diagrama do multiciclo <i>RV32I e RV32IM</i> .....	56
II.4	Diagrama do multiciclo <i>RV32IMF</i> .....	57
II.5	Diagrama do <i>pipeline RV32I e RV32IM</i> .....	58
II.6	Diagrama do <i>pipeline RV32IMF</i> .....	59

# LISTA DE TABELAS

3.1	Tabela de <i>syscalls</i> implementadas .....	31
4.1	Características dos sistemas implementados .....	41
4.2	Comparativo de desempenho de cada <i>ISA</i> em microarquiteturas distintas .....	44



# LISTA DE SÍMBOLOS

## Siglas

ASIC	Circuito Integrado de Aplicação Específica — <i>Application Specific Integrated Circuit</i>
BSD	Distribuição de Software de Berkeley — <i>Berkeley Software Distribution</i>
CISC	Computador com Conjunto de Instruções Complexo — <i>Complex Instruction Set Computer</i>
CSR	Registradores de Controle e Estado — <i>Control and Status Registers</i>
DSP	Processamento Digital de Sinais — <i>Digital Signal Processing</i>
FPGA	Arranjo de Portas Programáveis em Campo — <i>Field Programmable Gate Array</i>
hart	<i>hardware thread</i>
HDL	Linguagem de Descrição de <i>Hardware</i> — <i>Hardware Description Language</i>
HLS	Síntese de Alto Nível — <i>High-Level Synthesis</i>
IDE	Ambiente Integrado de Desenvolvimento — <i>Integrated Development Environment</i>
ISA	Arquitetura do Conjunto de Instruções — <i>Instruction Set Architecture</i>
MIPS	Microprocessador sem Estágios Intertravados de <i>Pipeline</i> — <i>Microprocessor without Interlocked Pipeline Stages</i>
OAC	Organização e Arquitetura de Computadores
PC	Contador de Programa — <i>Program Counter</i>
PLL	Malha de Captura de Fase — <i>Phase-Locked Loop</i>
PCB	Placa de Circuito Impresso — <i>Printed Circuit Board</i>
RAS	Pilha de Endereços de Retorno — <i>Return Address Stack</i>
RISC	Computador com Conjunto de Instruções Reduzido — <i>Reduced Instruction Set Computer</i>
RTL	<i>Register-Transfer Level</i>
SBC	Computadores em Placa Única — <i>Single Board Computers</i>
SDK	Conjunto de Programas de Desenvolvimento — <i>Software Development Kit</i>
SSD	Unidade de Estado Sólido — <i>Solid State Driver</i>
SiMPLE	Ambiente de Aprendizado Uniciclo, Multiciclo e <i>Pipeline</i> — <i>Single-cycle Multicycle Pipeline Learning Environment</i>
SoC	Sistema em um Chip — <i>System on Chip</i>
TSMC	<i>Taiwan Semiconductor Manufacturing Company</i>

# Capítulo 1

## Introdução

Até recentemente, a disciplina de Organização e Arquitetura de Computadores da Universidade de Brasília era ministrada em todas as turmas utilizando a arquitetura *MIPS32*. Apesar da arquitetura *MIPS32* ainda ter grande força no meio acadêmico (em boa parte devido a sua simplicidade e extensa bibliografia), sua aplicação na indústria tem diminuído consideravelmente na última década.

### 1.1 Motivação

O mercado de trabalho está a cada dia mais exigente, sempre buscando profissionais que conheçam as melhores e mais recentes ferramentas disponíveis. Além disso, muitos universitários se sentem desestimulados ao estudarem assuntos desatualizados e com baixa possibilidade de aproveitamento do conteúdo no mercado de trabalho. Isso alimenta o desinteresse pelos temas abordados e, em muitos casos, leva à evasão escolar. Assim, é importante renovar as matérias com novas tecnologias e tendências de mercado sempre que possível, a fim de instigar o interesse dos discentes e formar profissionais mais capacitados e preparados para as demandas da atualidade.

Embora a curva de aprendizagem de linguagens *assembly* de alguns processadores *RISC* seja relativamente baixa para quem já conhece o *assembly MIPS32*, aprender uma arquitetura atual traz o benefício de conhecer o *estado da arte* da organização e arquitetura de computadores.

Hoje, a disciplina também é ministrada na arquitetura *ARM*, bem como na *ISA RISC-V*, desenvolvida na Divisão de Ciência da Computação da Universidade da Califórnia - Berkeley, e será o objeto de estudo desse trabalho.

### 1.2 Por que RISC-V?

A *ISA RISC-V* (lê-se “*risk-five*”) é uma arquitetura *open source* [1] com licença *BSD*, o que permite o seu livre uso para quaisquer fins, sem distinção de se o trabalho possui código-fonte aberto ou proprietário. Tal característica possibilita que grandes fabricantes utilizem a arquitetura

para criar seus produtos, mantendo a proteção de propriedade intelectual sobre seus métodos de implementação e quaisquer subconjuntos de instruções não-*standard* que as empresas venham a produzir, o que estimula investimentos em pesquisa e desenvolvimento.

Empresas como Google, IBM, Nvidia, Samsung, Qualcomm e Western Digital são algumas das fundadoras e investidoras da *RISC-V Foundation*, órgão responsável pela governança da arquitetura. Isso demonstra o interesse das gigantes do mercado em seu sucesso e disseminação.

A licença também permite que qualquer indivíduo produza, distribua e até mesmo comercialize sua própria implementação da arquitetura sem ter que arcar com *royalties*, sendo ideal para pesquisas acadêmicas, *startups* e até mesmo *hobbyistas*.

O conjunto de instruções foi desenvolvido tendo em mente seu uso em diversas escalas: sistemas embarcados, *smartphones*, computadores pessoais, servidores e supercomputadores, o que pode viabilizar maior reuso de *software* e maior integração de *hardware*.

Outro fator que estimula o uso do *RISC-V* na disciplina é a modernização dos livros didáticos. A nova versão do livro utilizado em OAC, Organização e Projeto de Computadores, de David Patterson e John Hennessy, utiliza a *ISA RISC-V*. Além disso, com a promessa de se tornar uma das arquiteturas mais utilizadas nos próximos anos, a escolha do *RISC-V* para a disciplina de OAC se mostra ideal no momento.

### 1.3 Objetivos

O projeto *RISC-V SiMPLE (Single-cycle Multicycle Pipeline Learning Environment)* consiste no aprimoramento e documentação do processador com conjunto de instruções *RISC-V*, sintetizável em *FPGA* e com *hardware* descrito em *Verilog* utilizado como material de laboratório de uma das turmas da disciplina de OAC. O objetivo é ter uma plataforma de testes e simulação bem documentada e para servir de referência na disciplina. O projeto implementa três microarquiteturas que podem ser escolhidas a tempo de síntese: uniciclo, multiciclo e *pipeline*, todas as três com um *hart* e caminho de dados de 32 bits.

Os processadores contém o conjunto de instruções I (para operações com inteiros) e as extensões M (para multiplicação e divisão de inteiros) e F (para ponto flutuante com precisão simples conforme o padrão IEEE 754 - 2008). A síntese das extensões M e F é opcional, sendo que ao optar pelo uso da extensão F, a M também é incluída. Assim, pela nomenclatura da arquitetura, os processadores desenvolvidos são chamados de *RV32I*, *RV32IM* ou *RV32IMF*. Todas as versões implementam exceções e chamadas de sistema (*syscalls*) [2].

### 1.4 Estrutura do texto

A monografia possui cinco capítulos, sendo eles a presente introdução, um capítulo dedicado a revisar os assuntos pertinentes ao trabalho, o seguinte para descrever o sistema proposto, o penúltimo para apresentar os resultados obtidos e o último para as ponderações finais.

# Capítulo 2

## Revisão Teórica

Alguns conceitos são essenciais para compreender o projeto proposto, sua implementação e seus resultados. Esse capítulo abordará o que é arquitetura e organização de computadores, tratará brevemente sobre as arquiteturas mais conhecidas e se aprofundará na especificação da arquitetura *RISC-V*. Além disso, serão expostos os conceitos de síntese de *hardware* e do funcionamento das *FPGAs*, em especial o modelo que será utilizado para desenvolver o trabalho. Por fim, o “estado da arte” da arquitetura *RISC-V* será discutido.

### 2.1 Arquitetura de Computadores

Para nos comunicarmos, necessitamos de uma linguagem, e no caso dos brasileiros, essa linguagem é o português. Como toda linguagem, o português possui sua gramática e dicionário que lhe dá estrutura e sentido. Línguas humanas como o português, inglês e espanhol são chamadas de linguagens naturais, e evoluíram naturalmente a partir do uso e repetição. [3]

Por causa da excelente capacidade de interpretação e adaptação da mente humana, somos capazes de criar e entender novos dialetos que não seguem as regras formais das linguagens naturais que conhecemos. Porém, fora da comunicação casual é importante e às vezes obrigatório que nos expressemos sem ambiguidade. Línguas artificiais como a notação matemática e linguagens de programação possuem semântica e sintaxe mais rígidas para garantir que a mensagem transmitida seja interpretada da maneira correta. Sem essa rigidez, os computadores de hoje não seriam capazes de entender nossos comandos.

Para a comunicação com o processador de um computador, utilizamos mensagens chamadas de instruções, e o conjunto dessas instruções é chamado de Arquitetura do Conjunto de Instruções (*ISA*). Um processador só é capaz de entender as mensagens que obedecem as regras semânticas e sintáticas de sua *ISA*, e qualquer instrução que fuja das suas regras causará um erro de execução ou realizará uma tarefa diferente da pretendida. A linguagem de máquina é considerada de baixo nível pois apresenta pouca ou nenhuma abstração em relação à arquitetura.

As instruções são passadas para o processador na forma de código de máquina, sequências de

dígitos binários que correspondem aos níveis lógicos do circuito. Para melhorar o entendimento do código e facilitar seu desenvolvimento, uma outra representação é utilizada, o *assembly*. Um código *assembly* é transformado em código de máquina por um programa montador (*assembler*), e o processo inverso é realizado por um *disassembler*. As linguagens *assembly*, dependendo do *assembler* utilizado, permitem o uso de *macros* de substituição e pseudo-instruções (determinadas instruções inexistentes na *ISA* que são expandidas em instruções válidas pelo montador) e são totalmente dependentes da arquitetura do processador, o que normalmente impede que o mesmo código binário executável funcione em arquiteturas diferentes.

A Figura 2.1 é uma representação simplificada de um processador. A unidade de controle lê uma instrução da memória e a decodifica; o circuito de lógica combinacional lê os dados dos registradores, entrada e memória conforme necessário, executa a instrução decodificada e escreve no banco de registradores, na memória de dados ou na saída se for preciso; a unidade de controle lê uma nova instrução e o ciclo se repete até o fim do programa. A posição de memória da instrução que está sendo executada fica armazenada em um registrador especial chamado de Contador de Programa (*PC*). Algumas instruções modificam o *PC* condicionalmente ou diretamente, criando a estrutura para saltos, laços e chamada/retorno de funções.

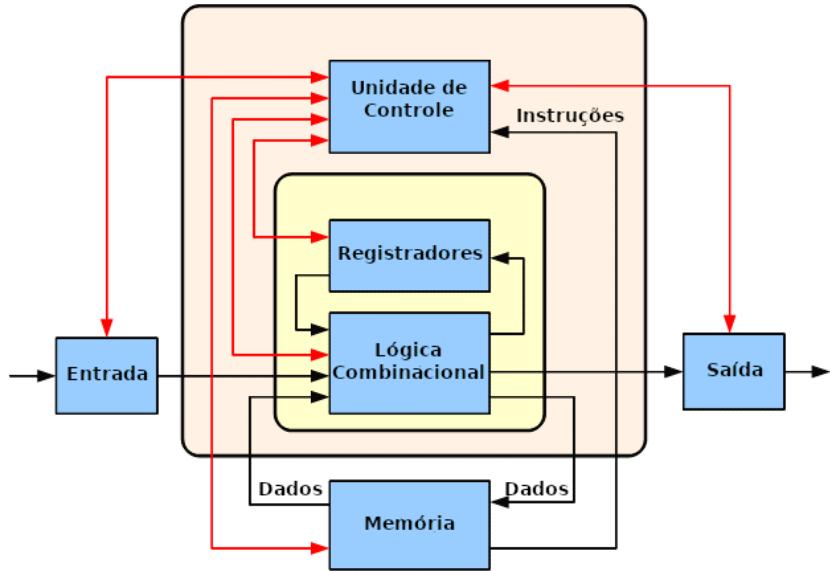


Figura 2.1: Abstração da arquitetura de um computador. Fonte: [4]

### 2.1.1 Arquiteturas *RISC* e *CISC*

Historicamente, as arquiteturas são divididas em *ISAs RISC* (*Reduced Instruction Set Computer*) e *CISC* (*Complex Instruction Set Computer*). Na atualidade, a principal diferença entre elas é que as *ISAs RISC* acessam a memória por instruções de *load/store*, enquanto as *CISC* podem

acessar a memória diretamente em uma instrução de operação lógica ou aritmética. [5]

Algumas arquiteturas *RISC* notáveis são a *RISC-V*, objeto de estudo desse trabalho, a *ARM* e a *MIPS*. Quanto às *CISC*, a *x86* e sua extensão de 64 bits, a *AMD64*, são as mais conhecidas e amplamente utilizadas.

### 2.1.2 Arquitetura MIPS

Durante seus 36 anos de existência, a *ISA MIPS* (*Microprocessor without Interlocked Pipelined Stages*) teve diversas versões produzidas, atendendo às demandas de diversos segmentos do mercado. Usado em supercomputadores na década de 90 e nos famosos *videogames Nintendo 64*, *PlayStation*, *PlayStation 2* e *PlayStation Portable*, a arquitetura atendia a vários segmentos do mercado.

A *MIPS32*, uma de suas versões lançada em 1999 [6], é amplamente utilizada no meio acadêmico, utilizando livros como as diversas edições do *Computer Organization and Design* [7] e o clássico *See MIPS run* [8] como livros-texto das disciplinas de arquitetura de computadores.

Na última década, o uso de processadores *MIPS* têm se limitado a sistemas automotivos e roteadores de *internet* [9], e a empresa detentora dos direitos da arquitetura anunciou o fim do desenvolvimento de novas iterações. [10]

### 2.1.3 Arquitetura ARM

A arquitetura *ARM* (*Advanced RISC Machines*) é considerada extremamente eficiente no consumo de energia e possui boa dissipação de calor sem comprometer seu desempenho [11], e com isso é predominante no mercado de *smartphones* [12]. Presente também nos *videogames Nintendo 3DS* e *Nintendo Switch*, nos *Single Board Computers (SBCs) Raspberry Pi*, entre outros dispositivos, a arquitetura tem enorme força em aplicações embarcadas.

Porém, isso não a impede de atender a outros setores do mercado. Atualmente, o supercomputador mais potente do mundo utiliza a arquitetura *ARM*. [13] A *Apple* recentemente lançou seus primeiros *notebooks* e *desktops* com seu processador *Apple M1*, considerado hoje o processador de computadores pessoais com maior eficiência por watt. [14] O serviço de computação em nuvem *Amazon Web Services (AWS)* também oferece máquinas equipadas com seu processador *AWS Graviton* para computação geral.

A empresa *ARM* opera licenciando o uso do seu conjunto de instruções e de projetos de implementação, mas não produz *chips*. A manufatura do processador fica a cargo da empresa licenciante, como a *Qualcomm*, *NXP*, *Samsung* e *Apple*.

Por ser uma das arquiteturas prevalentes da atualidade, a *ISA ARM* era uma forte candidata para ser utilizada no presente trabalho. Porém, a *ISA ARM* não era uma arquitetura aberta, sendo necessário obter uma licença para desenvolvimento e distribuição de implementações. Atualmente, algumas de suas versões foram parcialmente abertas [15].

### 2.1.4 Arquitetura x86

A *ISA x86*, também conhecida como *IA-32* é uma arquitetura *CISC* de 32 bits criada em 1985 pela *Intel* para uso no seu processador *80386*, renomeado para *i386*. Um diferencial da *ISA x86* é que versões mais novas do conjunto de instruções mantém a compatibilidade com as versões anteriores. Assim, programas desenvolvidos para versões mais antigas continuam funcionando em sistemas modernos. Essa característica fez com que processadores *x86* dominassem o mercado de computadores pessoais e *workstations*.

Apesar de criada pela *Intel* para a manufatura de seus próprios processadores, a *ISA* foi licenciada para outras empresas como a *AMD*.

### 2.1.5 Arquitetura AMD64

A arquitetura *AMD64* também conhecida como *x86-64* é a extensão de 64 bits da *IA-32*, criada pela *AMD* em 1999. A *Intel* criou sua própria extensão para a *IA-32*, a *IA-64* da linha de processadores *Itanium*. Porém, a extensão criada pela *AMD* teve mais tração e a *Intel* também adotou a *AMD64*. A característica de iterações mais novas manterem compatibilidade com softwares desenvolvidos para versões anteriores também faz parte da especificação da arquitetura.

Hoje, a maioria predominante dos computadores pessoais modernos, *workstations* e servidores utilizam processadores *x86-64*.

## 2.2 A Arquitetura RISC-V

Conforme descrito na Seção 1.2, a *RISC-V* é uma arquitetura de especificação aberta e licença livre, o que permite que qualquer indivíduo ou entidade produza, distribua e/ou comercialize processadores que a utilizem sem haver cobrança de *royalties*.

A *RISC-V* é uma arquitetura modular, sendo o módulo base de operações com inteiros mandatório em qualquer implementação. Esse módulo será o **I** (de *Integer*) para a maioria das implementações. Ele possui 32 registradores, sendo 31 de uso geral e um *hardwired* para o valor **zero**. Mas também existe o módulo **E** (de *Embedded*) para aplicações embarcadas, que possui metade dos registradores presentes nas outras implementações (15 de uso geral mais o **zero**) e é limitado a registradores de 32 bits de largura. As demais extensões são de uso opcional.

O módulo **I** pode ser implementado com registradores de 32, 64 ou 128 bits de largura. O módulo com registradores de 128 bits é um planejamento para o futuro quando 64 bits não forem mais suficientes para o endereçamento de memória do sistema, fazendo com que a arquitetura não tenha que ser reprojetada para se adequar às novas demandas. Assim, os possíveis conjuntos de instrução base são: *RV32I*, *RV32E*, *RV64I* e *RV128I*.

O *design* do módulo visa reduzir o *hardware* necessário para uma implementação mínima, bem como ser um alvo de compilação satisfatório, sendo capaz de emular por *software* as extensões **M** (de *Multiplication and Division*), **F** (de *Single-Precision Floating-Point*) e **D** (de *Double-Precision*

*Floating-Point*). Para isso, são implementadas instruções de *load/store* para transferir dados entre a memória e os registradores, operações lógicas e aritméticas entre dados de registradores, instruções para transferência de controle (*jumps* e *branches* condicionais) e chamadas de sistema (*syscalls*).

Diferente de outras arquiteturas como a *ARM*, as instruções de multiplicação e divisão não fazem parte do conjunto básico, uma vez que necessitam de circuito especializado e por isso encarecem o desenvolvimento e produção dos processadores.

A arquitetura foi projetada para aceitar instruções de tamanho variável. A codificação do tamanho das instruções é mostrada na Figura 2.2. As instruções do módulo **I** seguem o *encoding* de instruções de 32 bits. A extensão **C** (de *Compact*) especifica instruções utilizando a codificação de 16 bits. As demais extensões previstas pela especificação da arquitetura também utilizam a codificação de 32 bits. As codificações para instruções com mais de 32 bits de largura foram projetadas para acomodar necessidades futuras e para permitir que projetistas extendam a arquitetura com módulos proprietários.

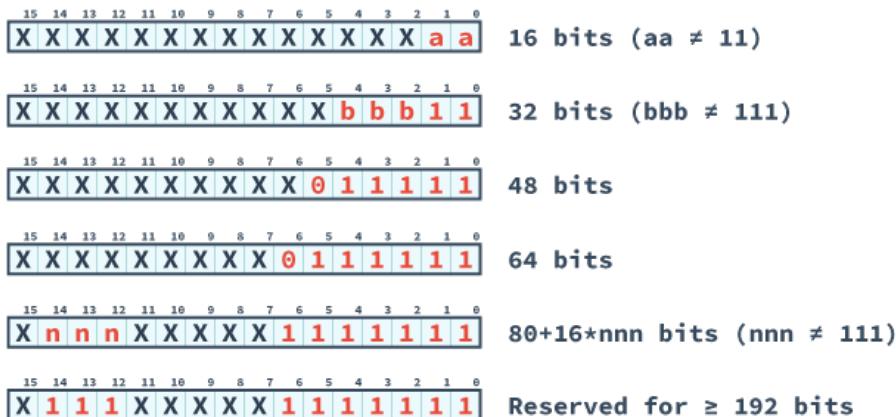


Figura 2.2: Codificação de instruções de tamanho variável da arquitetura *RISC-V*

### 2.2.1 Extensão de Registradores de Controle e Estado

A extensão **Zicsr** implementa o banco de *Control and Status Registers (CSRs)* e as instruções para acessá-los. Esses registradores contemplam um relógio de tempo real (*RTC*), contadores de instruções processadas e *ticks* do *clock*, além de habilitar/desabilitar funcionalidades do processador e checar seu estado.

### 2.2.2 Extensão de Multiplicação e Divisão

As operações aritméticas de multiplicação, divisão e resto de números inteiros são implementadas pela extensão **M**.

### 2.2.3 Extensões de Ponto Flutuante

A extensão **F** implementa instruções de ponto flutuante *IEEE 754 - 2008* de precisão simples que adiciona no processador um banco de 32 registradores de 32 *bits* usado para suas operações e um *CSR* para controlar o modo de arredondamento e identificar anormalidades na execução de uma instrução e.g. divisão por zero.

Suas instruções contemplam *load/store* no banco de registradores de ponto flutuante, transferência de dados entre os registradores de ponto flutuante e os de números inteiros (com ou sem conversão para inteiro), operações aritméticas e comparações.

A extensão **D** requer a implementação da extensão **F** e adiciona suporte para operações de ponto flutuante de precisão dupla, aumentando a largura do banco de registradores de ponto flutuante para 64 *bits*. Há ainda a extensão **Q** para operações de ponto flutuante com precisão quádrupla, que depende da extensão **D** estar implementada e extende a largura do banco de registradores de ponto flutuante para 128 *bits*.

### 2.2.4 Extensões A e Zifencei

A extensão **A** implementa instruções de acesso atômico a memória, enquanto a extensão **Zifencei** implementa *fencing* para escrita e leitura da memória de instruções em um *hart*.

### 2.2.5 Nomenclatura da implementação

A nomenclatura de um processador *RISC-V* segue a seguinte estrutura: as letras **RV** seguidas da largura do banco de registradores (**32/64**), da letra do módulo base (**I/E**), conforme visto na Seção 2.2, e em seguida as outras extensões implementadas.

As extensões **M**, **A**, **F**, **D**, **Zicsr** e **Zifencei** utilizadas com o módulo **I** constituem um conjunto considerado um alvo razoável para o desenvolvimento de um processador de uso geral. Pela convenção de nomenclatura, um processador de 64 *bits* com essa implementação se chamaría **RV64IMAFDZicsr\_Zifencei**. Para uma melhor apresentação, o conjunto **IMAFD-Zicsr\_Zifencei** pode ser abreviado para **G** (de *General-purpose*). Assim, o processador descrito pode ser chamado de **RV64G**.

### 2.2.6 Outras Extensões

Existem propostas de adição de algumas extensões na especificação da *ISA*, como a **B** para manipulação de *Bits* e a **V** para operar *Vetores*, mas ainda não foram ratificadas.

Há também a possibilidade de se desenvolver extensões próprias. A regra de nomenclatura para esse caso é iniciar o nome da extensão com um **X** seguido por um nome alfabético e.g. **Xhwacha**.

## 2.2.7 Arquitetura Privilegiada

Para a *ISA RISC-V*, existem quatro níveis de privilégio de acesso, sendo eles: o de máquina (nível *bare-metal* que em implementações mais simples trata *syscalls*), o de usuário (o nível onde a aplicação do usuário é executada), o de supervisor (nível do sistema operacional) e o de hipervisor (virtualização).

O nível privilegiado de máquina adiciona alguns *CSRs* importantes como o `misa` (*Machine ISA register*), que indica as capacidades do processador como largura dos registradores e extensões implementadas codificados em *bit fields*, o `mstatus` que controla o estado operacional do *hart* como a habilitação de interrupções, o `mtvec` que contém o endereço de memória do tratamento de *traps*, o `mepc` que guarda o endereço de memória onde uma instrução causou uma exceção e o `mcause` que indica a causa da exceção.

## 2.2.8 Formatos de Instruções

As instruções da arquitetura podem ser separadas em subgrupos de acordo com os operadores necessários para o processador interpretá-la. A Figura 2.3 apresenta os formatos das instruções do módulo I da *ISA RISC-V*, e, para efeitos de comparação, a Figura 2.4 mostra os formatos de instruções equivalentes na arquitetura *MIPS32*.

Apesar dos formatos das instruções da *RISC-V* serem mais complicados que os da *MIPS32* do ponto de vista de um humano, para uma máquina eles são muito mais eficientes. Os campos do `opcode`, do `funct3`, dos registradores `rs1`, `rs2` e `rd`, e alguns bits dos imediatos `imm` sempre se encontram na mesma posição na instrução. Isso reduz a complexidade dos multiplexadores de geração de imediatos e da lógica de decodificação da instrução. Além disso, por utilizar imediatos de 12 *bits* (ou 20 no caso das instruções do tipo **U**) em vez de imediatos de 16 *bits* (ou 28 nas instruções do tipo **J**), a *ISA RISC-V* possui mais *bits* disponíveis para codificar mais instruções.

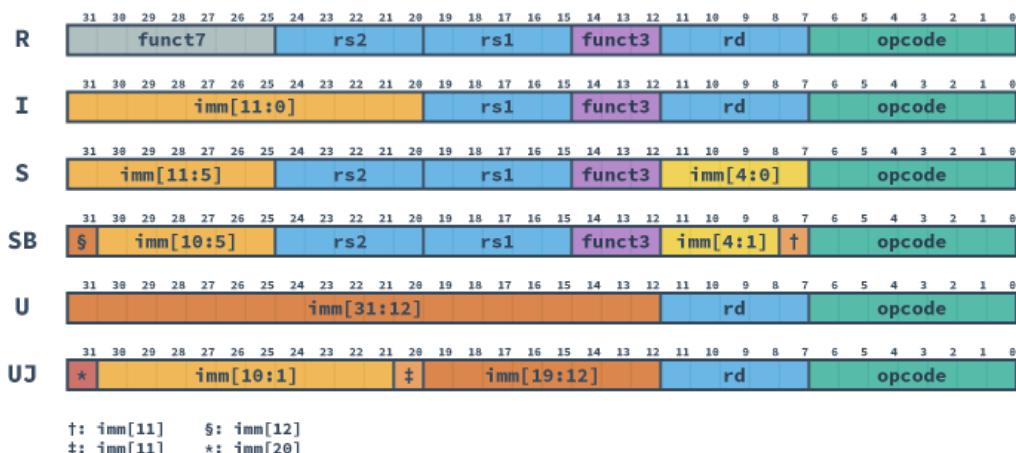


Figura 2.3: Formatos de Instruções da *ISA RISC-V*

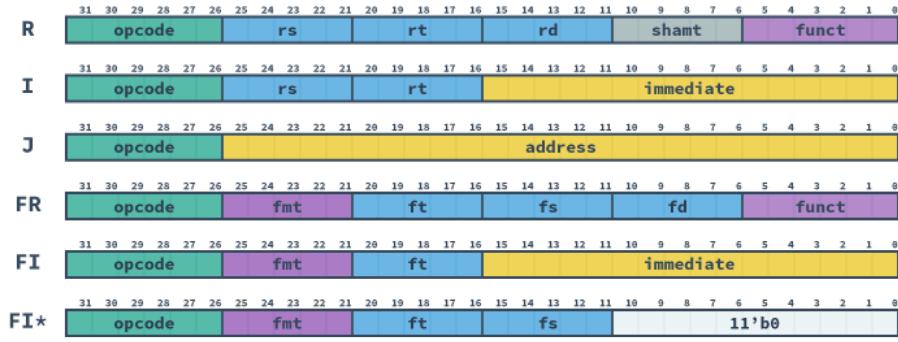


Figura 2.4: Formatos de Instruções da *ISA MIPS32*

### 2.2.9 Formatos de Imediatos

Os imediatos são operandos descritos na própria instrução em vez de estarem contidos em um registrador. Como os operandos necessitam ter a mesma largura que o banco de registradores, algumas regras são utilizadas para gerar os imediatos. As figuras a seguir mostram a formação de cada tipo de imediato dos formatos das instruções apresentadas na Figura 2.3.

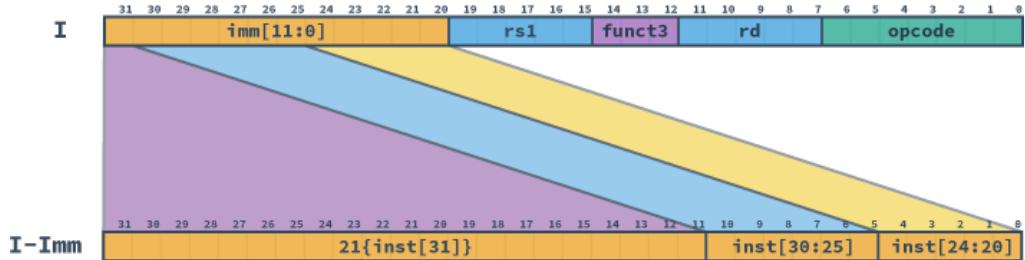


Figura 2.5: Formação do Immediato de tipo I

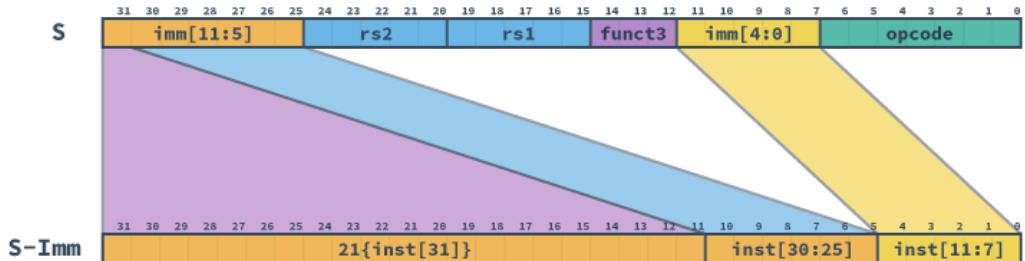


Figura 2.6: Formação do Immediato de tipo S

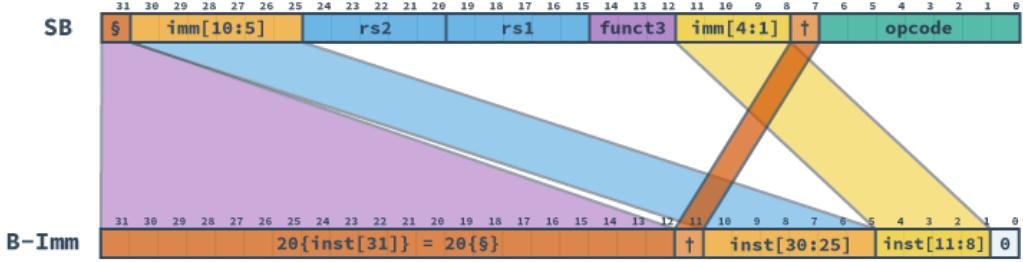


Figura 2.7: Formação do Imediato de tipo B

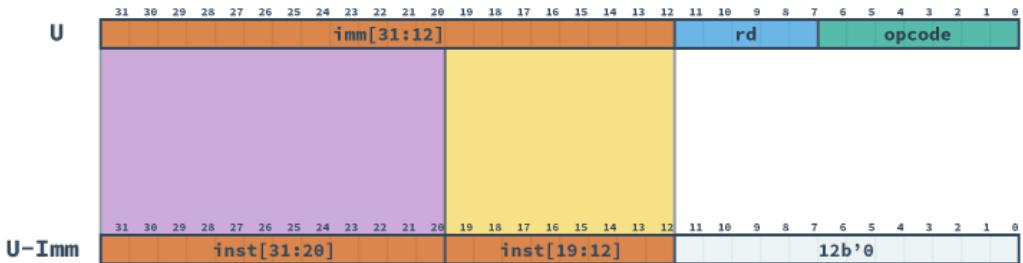


Figura 2.8: Formação do Imediato de tipo U

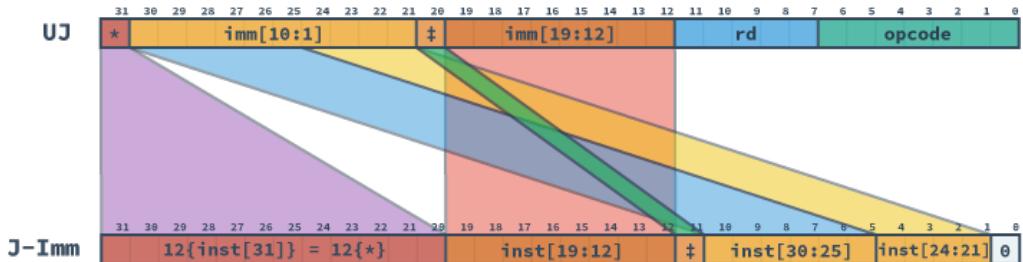


Figura 2.9: Formação do Imediato de tipo J

## 2.3 IDE RARS

O Ambiente Integrado de Desenvolvimento (*IDE*) *RARS* (*RISC-V Assembler and Runtime Simulator*) é um *software open source* [16] utilizado para desenvolver, montar, simular e exportar código em *assembly RISC-V*. O projeto é um *fork* do *MARS* (*MIPS Assembler and Runtime Simulator*), outro *software open source* [17] muito utilizado em cursos de arquitetura de computadores ministrados em *MIPS32*.

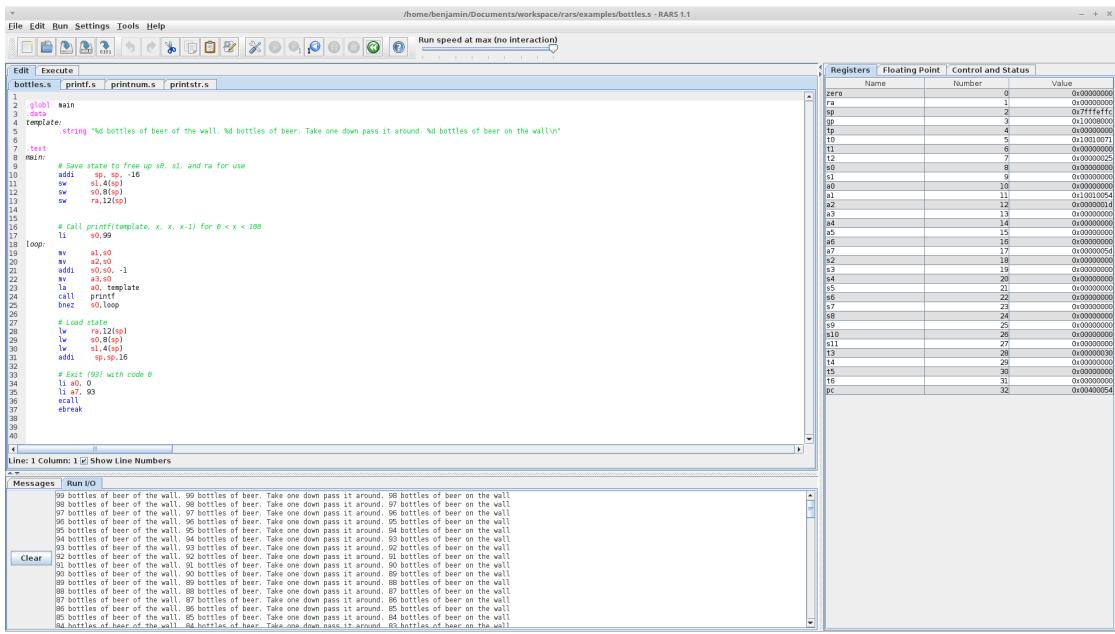


Figura 2.10: *IDE RARS*

O *RARS* é compatível com as *ISAs* **RV32IMFD** e **RV64IMFD** com suporte a interrupções e implementa parcialmente a extensão **Zicsr**. Também tem suporte a *syscalls*. Possui também um guia de programação em *assembly RISC-V* e pode ser utilizado como referência de instruções e pseudo-instruções.

Com sua função de simulador, permite executar o código instrução por instrução, mostrando as modificações nos bancos de registradores e na memória. Além disso, permite exportar o programa montado no formato *.mif* para uso na inicialização de memória de *FPGAs*.

## 2.4 Microarquiteturas

Enquanto o campo de arquitetura de computadores explora a especificação dos conjuntos de instruções, a organização de computadores trata da forma com que as especificações são implementadas. Os padrões de implementação de *ISAs* são chamados de microarquiteturas, e esses padrões podem ser combinados para produzir soluções mais complexas.

Processadores são circuitos elétricos, e podem ser descritos por diagramas. Nesses diagramas, o caminho de dados (*datapath*) representa a conexão feita entre blocos lógicos por barramentos onde os sinais a serem processados trafegam. A unidade de controle é responsável por interpretar as instruções recebidas e comandar os blocos lógicos para executar o comando. Os blocos lógicos são as Unidades de Lógica e Aritmética (ULA, ou *ALU* em inglês), bancos de registradores (*regfiles*) multiplexadores e blocos de memória.

### 2.4.1 Uniciclo

Processadores uniciclo são implementados de forma com que cada instrução seja recuperada (*fetch*), decodificada (*decode*), executada e finalizada (*retire*) durante um único *tick* do *clock*. Para isso, a unidade de controle deve conter somente lógica combinacional e a frequência de operação do *clock* deve ser projetada para o pior caso, ou seja, a instrução que leva mais tempo entre seu *fetching* e seu *retiring*.

A implementação da microarquitetura uniciclo é uma das mais simples de se fazer, mas o processador resultante apresenta baixa frequência de processamento. Para *workloads* que utilizam muitas instruções complexas, seu desempenho pode ser parecido ou até melhor que implementações mais elaboradas, mas se a maioria da carga de processamento for de instruções rápidas, o uniciclo sofre uma penalidade por ficar ocioso durante parte do seu ciclo.

### 2.4.2 Multiciclo

A microarquitetura multiciclo também é conhecida como implementação por microcódigo. As instruções são “quebradas” em etapas menores, como o acesso à memória, a leitura do banco de registradores, a seleção de um sinal multiplexado, a realização de uma operação lógica ou aritmética, dentre outras possibilidades. A unidade de controle é implementada por lógica sequencial como uma máquina de estados finitos, e a frequência máxima de operação é limitada pela etapa mais lenta da máquina de estados.

Como a quantidade de estados que uma instrução deve passar entre seu *fetching* e *retiring* depende de sua complexidade e de como a máquina de estados foi projetada, a frequência do *clock* costuma ser mais alta que de uma implementação uniciclo, mas isso não significa que seu desempenho será melhor, pois o desempenho também depende do *workload*. Se a maioria das instruções executadas precisa passar por diversos estados para serem completadas, o somatório de ciclos menores pode resultar em um intervalo de tempo maior que o do ciclo único.

### 2.4.3 Pipeline

Uma implementação *pipeline* consiste em inserir registradores entre os blocos lógicos do *data-path*, o dividindo em etapas. O objetivo é ser capaz de processar mais de uma instrução por vez. A Figura 2.11 mostra um *pipeline* genérico de quatro estágios. O processo inicia com o *fetching* de uma instrução no primeiro ciclo do *clock*. No segundo ciclo do *clock*, a primeira instrução passa para o estágio de *decode* e uma nova instrução entra no primeiro estágio do *pipeline*. No terceiro *tick*, a primeira instrução está no terceiro estágio, a segunda no segundo estágio e a terceira no primeiro estágio. No quarto ciclo, a primeira instrução está no quarto estágio, a segunda instrução no terceiro estágio, a terceira instrução no segundo estágio e a quarta instrução acaba de entrar no *pipeline*. No ciclo seguinte, a primeira instrução é finalizada, as outras três instruções passam para o estágio seguinte e uma nova instrução entra no primeiro estágio do *pipeline*.

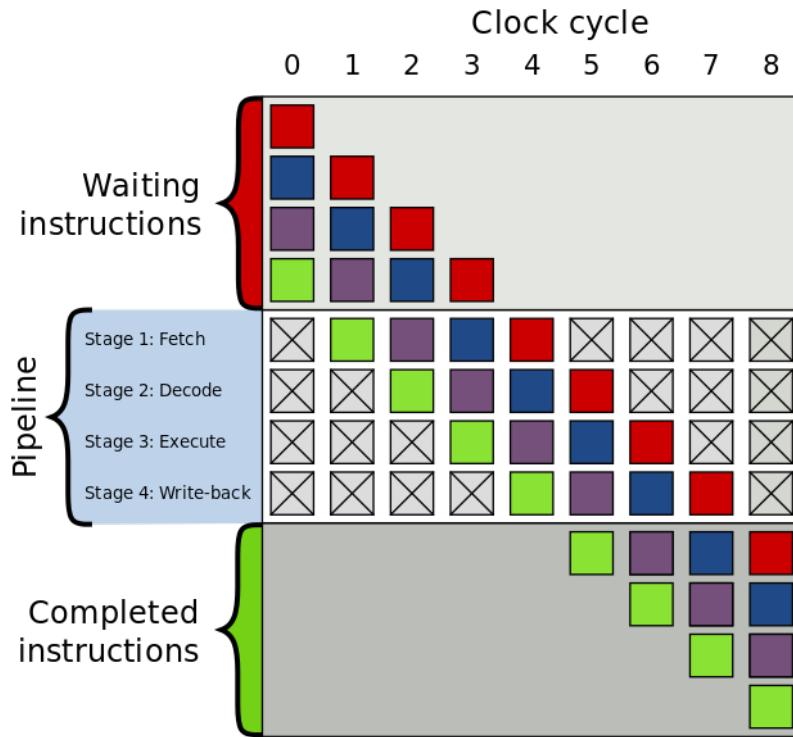


Figura 2.11: *Pipeline* genérico com 4 estágios.    Fonte: [18]

A frequência do *clock* é limitada pelo estágio mais lento. O objetivo é manter o *pipeline* com todos os seus estágios preenchidos, e assim, a cada *tick*, uma instrução é completada, sendo uma implementação com ótimo desempenho. Porém, é comum que uma instrução precise do resultado de outra instrução que ainda não foi finalizada.

Uma solução simples porém ineficiente para o problema é limpar os estágios anteriores, inserindo *nops* (instruções que não realizam nenhuma tarefa) até que o resultado esteja disponível e voltar a preencher o *pipeline*. Uma solução mais robusta seria implementar uma unidade de identificação e tratamento de *forwards* e *hazards*. *Forwards* ocorrem quando a instrução anterior ainda não finalizou, mas seu resultado já está disponível em algum estágio do *pipeline* e o valor pode ser injetado na etapa que precisa dele. Já os *hazards* são situações onde o *forward* não é possível e *nops* precisam ser inseridos.

## 2.5 Representação de *Hardware*

Existem diversas maneiras de representar circuitos eletrônicos. Na programação visual, as entradas e saídas dos blocos funcionais são conectados de forma gráfica, montando um diagrama do circuito. Já as linguagens de descrição de *hardware* (*HDLs*) como o *Verilog* e o *VHDL* permitem uma descrição precisa da estrutura e comportamento dos circuitos. Outra forma de representação utiliza a síntese de alto nível (*High-Level Synthesis*), onde se desenvolve um algoritmo em linguagens como *C++* ou *MATLAB*, que será “traduzido” para um circuito eletrônico.

Essas representações permitem a geração de *netlists* utilizadas para simular o projeto, sintetizá-lo em *FPGAs* ou fazer o *tapeout* para produção de *chips*.

### 2.5.1 Verilog

*Verilog* é uma *HDL* comumente utilizada no projeto *RTL* de circuitos lógicos. Módulos escritos em *Verilog* descrevem seus sinais de entrada e saída como parâmetros para a conexão com outros módulos. As suas principais “variáveis” são *wires* (fios que conectam um sinal a outro) e *regs* (registradores ou *drivers* de sinal). A lógica do módulo pode ser implementada por *assignments* (a conexão de um fio a um registrador ou um *driver*) ou por blocos de lógica sequencial ou combinacional, usando operadores blocantes (=) ou não-blocantes (<=).

Para quem está acostumado com linguagens imperativas como o *C*, há uma barreira inicial para entender que o código não é executado linha por linha, mas sim de maneira concorrente, e que a escolha entre operadores blocantes e não-blocantes afeta a forma com que os sinais são propagados no circuito.

A sintaxe do *Verilog* é menos verbosa que a do *VHDL*, que é fortemente tipada. Enquanto a verificação formal de *designs* em *VHDL* é um processo mais natural, o *Verilog* possui uma curva de aprendizado menor, sendo ideal para o primeiro contato com linguagens de descrição de *hardware*.

### 2.5.2 Análise e Síntese

O processo de análise de representações de *hardware* verifica a sintaxe da representação, identificando estruturas inválidas, sinais não conectados e possíveis erros semânticos que farão com que a *netlist* não possa ser gerada ou não se comporte conforme o esperado.

Já a síntese é o processo de transformar a estrutura analisada em uma *netlist*, estrutura que contém a lista de componentes eletrônicos do *design* e a descrição de como eles se conectam.

### 2.5.3 Fitting e Timing Analyzer

O *fitting* é o processo de criar as rotas de conexão da *netlist*. O *design* é mapeado para a plataforma onde será implementado. Já o *timing analyzer* define restrições de temporização entre elementos do circuito que o *fitter* tenta respeitar. O *fitter* itera diversas vezes o posicionamento e roteamento dos componentes até gerar o resultado final.

Com a *netlist* roteada, o *Timing Analyzer* é executado novamente, testa se o *fitter* conseguiu atender às restrições e gera um relatório. Caso alguma restrição essencial não for respeitada, o *Fitter* pode ser novamente executado com opções diferentes que impactarão outras características do circuito final, como quantidade de componentes e consumo energético.

Pode ser necessário reimplementar a representação do *hardware* para que todos os requisitos de projeto sejam atendidos.

## 2.6 Field Programmable Gate Arrays

*Field Programmable Gate Arrays (FPGAs)* são circuitos integrados que permitem o desenvolvimento de circuitos lógicos reconfiguráveis. Por serem reprogramáveis, as *FPGAs* geram uma grande economia em tempo de desenvolvimento e em custos como os de prototipagem, validação e manufatura do projeto em relação aos circuitos de aplicações específicas, os *ASICs*, e aos projetos *full-custom*. As *FPGAs* podem ser tanto o passo intermediário no projeto de um *ASIC* ou *full-custom* quanto o meio final do projeto quando a reconfigurabilidade e os preços muito mais acessíveis forem fatores importantes.

Cada fabricante de *FPGAs* possui seu *software* de desenvolvimento, ou *SDKs*. A indústria de *hardware* é extremamente protecionista com sua propriedade intelectual (tanto que os *designs* de módulos ou circuitos completos são chamados de *IPs*, termo derivado de *Intellectual Property*), sendo a maioria dessas ferramentas de código proprietário. Para a Intel Altera, essa plataforma é o Quartus Prime.

*FPGAs* mais modernas possuem, além do arranjo de portas lógicas, blocos de memória, *Phase-Locked Loops (PLLs)*, *Digital Signal Processors (DSPs)* e *System on Chips (SoCs)*. Os blocos de memória internos funcionam como a memória *cache* de um microprocessador, armazenando os dados próximo ao seu local de processamento para diminuir a latência. Os *PLLs* permitem criar sinais de *clock* com diversas frequências a partir de um relógio de referência, e podem ser reconfigurados a tempo de execução. *DSPs* são responsáveis pelo processamento de sinais analógicos discretizados, e podem ser utilizados como multiplicadores de baixa latência. Já os *SoCs* são microprocessadores como os ARM presentes em celulares, e são capazes de executar sistemas operacionais como o Linux.

Além de disponíveis na forma de *chips* para a integração com placas de circuito impresso customizadas (*PCBs*), as *FPGAs* possuem *kits* de desenvolvimento com diversos periféricos para auxiliar no processo de criação de soluções. Esses *kits* são a principal ferramenta de aprendizagem no universo dos circuitos reconfiguráveis. No Laboratório de Informática da UnB, as placas *terasIC DE1-SoC* com a *FPGA Intel Cyclone V SoC* estão disponíveis para os alunos de OAC desenvolverem seus projetos.

### 2.6.1 Arquitetura Generalizada de uma FPGA

De forma genérica, uma *FPGA* possui blocos lógicos, chaves de interconexão, blocos de conexão direta e portas de entrada e saída, conforme apresentado na Figura 2.12.

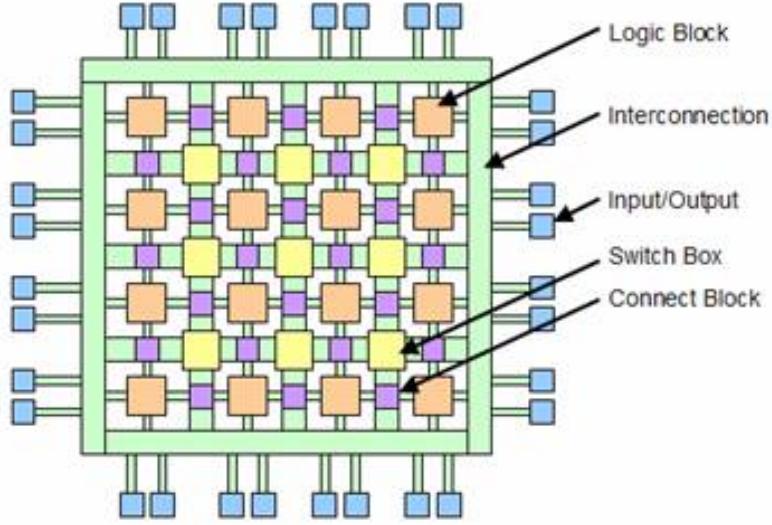


Figura 2.12: Abstração da arquitetura de uma FPGA. Fonte: [19]

Os blocos lógicos possuem *lookup tables*, registradores, somadores e multiplexadores. É neles que a lógica reconfigurável é implementada. Já as chaves de interconexão são responsáveis por conectar os diversos blocos lógicos da *FPGA*. A Figura 2.13 exemplifica como é feito o roteamento da malha de interconexão. Os blocos de conexão direta são um tipo especial de chave de interconexão, e sua função é ligar blocos lógicos adjacentes. Por fim, as portas de entrada e saída conectam a *FPGA* ao “mundo externo” e.g. *drivers* de áudio e vídeo.

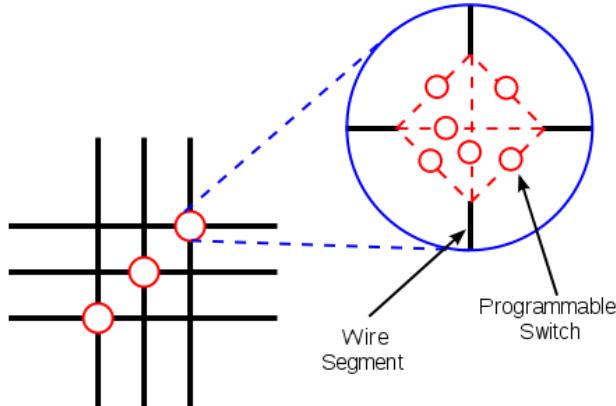


Figura 2.13: Funcionamento da chave de interconexão. Fonte: [20]

### 2.6.2 Arquitetura da FPGA Cyclone V SoC

A Figura 2.14 apresenta a arquitetura da *FPGA Cyclone V SoC*. O *chip* possui um processador *ARM* integrado, blocos de memória embutidos, *DSPs* para acelerar operações como multiplicação de números ou processamento de sinais genéricos, diversos pinos para integrar o *chip* a um projeto de circuito mais complexo, *PLLs* para gerar diversos sinais de *clock*, entre outras funcionalidades.

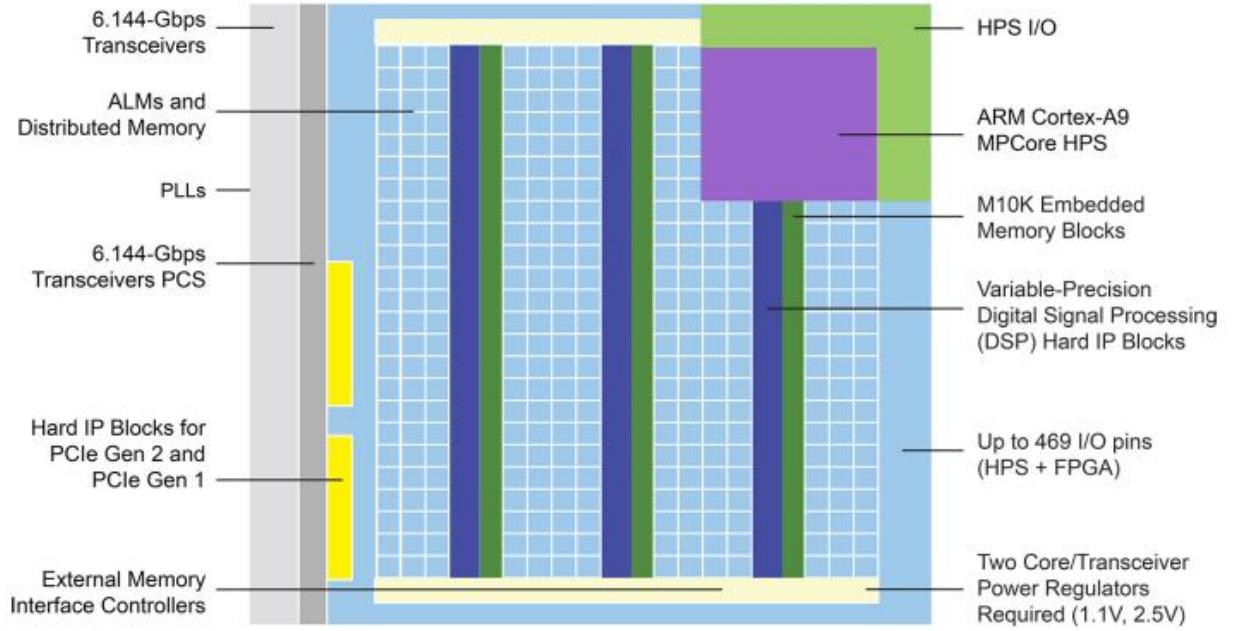


Figura 2.14: Arquitetura da *FPGA Altera Cyclone V SoC*. Fonte: [21]

### 2.6.2.1 Adaptative Logic Modules

Os blocos lógicos, como mostrados na abstração da Figura 2.12 são implementados na *FPGA Cyclone V SoC* como *Adaptative Logic Modules*, conforme a Figura 2.15. Como os *ALMs* são blocos genéricos, há um *trade-off* entre configurabilidade e performance.

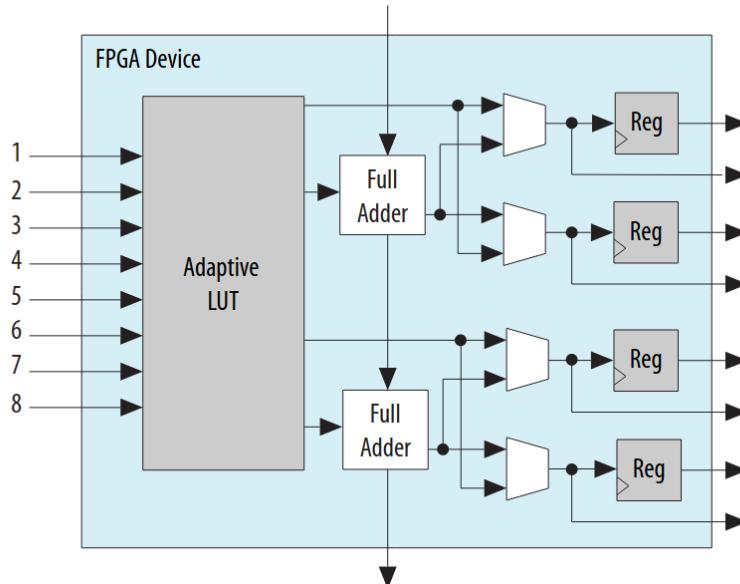


Figura 2.15: Diagrama de blocos de um ALM. Fonte: [22]

### 2.6.2.2 Embedded Memory Blocks

Como mostrado na Figura 2.14, existem blocos de memória embutidos na *FPGA*. Esses blocos são o equivalente a uma memória *cache L1*, sendo a camada de memória mais próxima dos registradores. Para utilizá-los no *design* do circuito, blocos do *IP* de memória são configurados e instanciados pelo programa de desenvolvimento para gerar um módulo integrável no resto do *design*.

As placas de desenvolvimento podem possuir outros tipos de memória, como as *SRAM* e *DRAM*. Apesar de possuírem capacidade de armazenamento bem maiores que os blocos embutidos, seus módulos controladores são mais complexos e apresentam latência maior para leitura e escrita de dados. Para seu uso eficiente, é necessário implementar camadas de *caching* para que as operações de *input* e *output* (*IO*) não se tornem um gargalo que comprometa o resto do *design*.

## 2.7 Estado da Arte dos processadores RISC-V

Por alguns anos, processadores com arquitetura *RISC-V* só podiam ser utilizados por meio de emuladores como o *qemu* [23] ou em *FPGAs*. Diversos *soft-cores open source*, tanto experimentais (como o desenvolvido neste trabalho) quanto de alto desempenho podem ser encontrados na *internet* e utilizados por qualquer pessoa interessada. Um dos processadores alto desempenho disponíveis é o *BOOM* [24], processador com execução *Out of Order* desenvolvido na universidade americana *UC Berkeley* pelo mesmo departamento onde surgiu a arquitetura *RISC-V*, o *Berkeley Architecture Research* [25]. O processador pode ser sintetizado em instâncias *EC2 F1* da *AWS* [26], eliminando a necessidade de possuir uma *FPGA* de alto desempenho para usufruir do projeto.

Outro projeto *open source* notável é o *Shakti* [27], ecossistema desenvolvido na universidade indiana *IIT-Madras* com diversas implementações para os mais diversos usos, que variam desde microcontroladores de 32 *bits*, superescalares de 64 *bits* para aplicação em *desktops* e servidores até processadores de alta performance com mais de 32 *cores* [28].

Porém, a presença apenas de *soft-cores* limita as possíveis aplicações da arquitetura. Algumas fabricantes já divulgaram planos para começar a usar microcontroladores com a arquitetura em seus produtos, como é o caso dos controladores de discos rígidos e *SSDs* da *Western Digital* [29] e da *Nvidia* como o substituto dos controladores *Falcon* de suas placas de vídeo [30]. Ainda não se sabe se as empresas já utilizam os controladores em seus *hardwares*, se a adoção ainda está em fase de projeto ou se a ideia foi abandonada.

Porém, começam a surgir no mercado microcontroladores e *Single Board Computers (SBCs)* com preços acessíveis. Placas como a linha *Sipeed* da *Seed* se equiparam aos *MCUs ESP32* [31], e outras como a *SiFive HiFive1* se assemelham aos *arduinoss* [32].

Há uma expectativa de *SBCs* mais robustos, capazes de rodar um sistema operacional de uso geral, como um *Raspberry Pi*. Existem alguns pré-lançamentos de placas para atender essa demanda, como a *SiFive HiFive Unmatched* [33] e a *BeagleV* [34].

A empresa *SiFive*, liderada pelos criadores da arquitetura, produzirá em parceria com a *TSMC* (*Taiwan Semiconductor Manufacturing Company*) o primeiro processador *RISC-V* de 32 bits em tecnologia de 5nm [35]. A *TSMC* é a *foundry* líder em manufatura de circuitos integrados no mundo.

Atualmente, compilar códigos em *C/C++* para *targets RISC-V* não envolve mais a instalação de *toolchains* complicadas e frágeis. Tanto o *gcc* [36] quanto o *clang* [37] já oferecem suporte para o *RISC-V*, eliminando assim uma barreira para a adoção da arquitetura.

Uma outra característica essencial para o uso do *RISC-V* em sistemas de uso geral é a existência de sistemas operacionais que funcionem na plataforma. Desde a versão 4.15, o *kernel* do *linux* oferece suporte para a arquitetura [38]. *Distros* como *Fedor*a [39], e *Alpine* [40] já possuem suporte experimental. A chinesa *Alibaba* fez o *port* do *OS Android* para um de seus *SoCs RISC-V* [41]. Alguns ecossistemas mais robustos possuem *ports* completos, como é o caso do *Haiku-OS* [42] e do *microkernel seL4* [43], possibilitando o uso em ambientes industriais e áreas que exigem maior robustez do sistema operacional.

Uma das surpresas na adoção da arquitetura *RISC-V* nos seus *designs* veio da *MIPS Technologies*, detentora das patentes das arquiteturas *MIPS*. Em 2013, a empresa foi adquirida pela *Imagination Technologies* [44], e lançou alguns *development kits* voltados a visão computacional e microcontroladores, mas não conseguiu dar tração aos projetos. Em 2017 a companhia foi novamente vendida para a *Tailwood Venture Capital* [45], que tentou capitalizar em cima dos *royalties* da arquitetura. Porém, em 2018 a companhia foi vendida novamente para a *Wave Computing* [46], companhia voltada para aplicações de inteligência artificial. Em 2020, a *Wave Computing* declara falência [10], demitindo todos os seus funcionários. Em março desse ano, a empresa conseguiu se recuperar da falência, mudou o nome da companhia para *MIPS* e anunciou que seus novos *designs* serão baseados na arquitetura *RISC-V* [47]. Atualmente, a empresa *MIPS* integra a *RISC-V Foundation* como Membro Estratégico.

## 2.8 Observações finais da Revisão Teórica

O capítulo abordou os conceitos que formam o alicerce do trabalho. O capítulo seguinte apresentará a proposta do projeto, descrevendo sua implementação e documentando o processo de execução e modificação do produto final a fim de permitir sua utilização futura.

# Capítulo 3

## Sistema Proposto

O sistema proposto consiste em um *soft-core* da *ISA RISC-V* de 32 *bits* com as extensões **I**, **M** e **F**, podendo ser sintetizado nas versões **RV32I**, **RV32IM** ou **RV32IMF**. A extensão **Zicsr** com os Registradores de Controle e Estado (*CSR*) é parcialmente implementada em todas as três configurações. Chamadas de sistema (*syscalls*) também são implementadas.

Cada uma das combinações da *ISA* pode ser realizada em três microarquiteturas diferentes: uniciclo, multiciclo ou *pipeline* de cinco estágios. Assim, o processador pode ser sintetizado em nove combinações diferentes.

O projeto utiliza a placa de desenvolvimento *terasIC DE1-SoC* contendo diversos periféricos e um *SoC Intel Altera Cyclone-V*. A maioria dos periféricos presentes na plataforma tem controladores implementados com Entradas e Saídas Mapeadas em Memória (*MMIO*) para que o *soft-core* possa utilizá-los. A síntese dos controladores dos periféricos, como a saída de vídeo, entrada de teclado e barramento *RS-232* é opcional.

### 3.1 Organização do projeto

O projeto é organizado seguindo o seguinte arranjo de pastas:

core	(arquivos que implementam o soft-core)
clock	(arquivos de interface e controle de sinais de clock do processador)
memory	(arquivos de interface/controle de memória)
misc	(módulos como somador e multiplexador de largura definidas por parâmetros)
peripherals	(interfaces e controladores para os periféricos da FPGA)
risc_v	(projeto do processador RISC-V)
CPU.v	(arquivo top-level do processador)
Control_*.v	(módulos de controle de cada microarquitetura)

<pre>         └── Datapath_*.v         └── ...     └── config.v     └── default_data.mif     └── default_framebuffer.mif     └── default_text.mif     └── fpga_top.sdc     └── fpga_top.v </pre>	<p>(módulos do caminho de dados de cada <code>param</code>)          (demais módulos do processador)</p> <p>(arquivo de configuração de versão do processador a implementar, seus periféricos e endereçamento de memória das interfaces MMIO)</p> <p>(arquivo de inicialização de memória de dados usado na síntese do projeto)</p> <p>(arquivo de inicialização de memória de vídeo usada na síntese do projeto)</p> <p>(arquivo de inicialização de memória de texto usado na síntese do projeto)</p> <p>(restrições desejadas de temporização do sistema sintetizado)</p> <p>(interface verilog entre o soft-core e a placa de desenvolvimento)</p>
<pre>     └── doc     └── project         └── de1_soc             └── db             └── incremental_db             └── output_files             └── fpga_top.qpf             └── fpga_top.qsf             └── ... </pre>	<p>(documentação e guias do projeto)</p> <p>(arquivos de projeto do Quartus)</p> <p>(arquivos de saída intermediários do Quartus; pasta ignorada pelo git)</p> <p>(arquivos de saída intermediários do Quartus; pasta ignorada pelo git)</p> <p>(arquivos de saída do Quartus; os logs de síntese gerados pelo script "make.sh" ficam aqui, bem como o .sof da última síntese completa; ignorada pelo git)</p> <p>(arquivo de projeto do Quartus indicando a versão do projeto)</p> <p>(arquivo de projeto do Quartus contendo as configurações do projeto)</p> <p>(outros arquivos de projeto do Quartus)</p> <p>(outros modelos de FPGA)</p>
<pre>     └── system     └── test         └── assembly_testbench         └── gtkwave </pre>	<p>(códigos em assembly RISC-V implementando as chamadas de sistema e macros)</p> <p>(códigos em assembly RISC-V para testar o funcionamento correto das instruções do processador)</p> <p>(formas de onda predefinidas para visualizar os arquivos .vcd gerados pelo ModelSim usando o GTKwave)</p>

```

    └── mif_library      (testbenches assembly compilados para o
                           formato .mif para gravação na memória
                           da FPGA)
    └── simulation       (arquivos de saída da simulação pelo
                           ModelSim; pasta ignorada pelo git)
    └── simulation_scripts (scripts .do para que o ModelSim simule o
                           sistema corretamente)
    └── sof_library      (arquivos .sof das versões do processador
                           prontos para gravação na FPGA)
    └── verilog_testbench (testbench usado para simular as entradas
                           da FPGA, inicializá-la e definir o
                           tempo de simulação)
  └── tools
    ├── bitmap_converter (conversor de imagens para uso na FPGA)
    ├── rars              (montador e simulador de assembly RISC-V)
    └── riscv-disassembler (disassembler de instruções usado para
                           traduzir o código de máquina dos
                           sinais de instruções no GTKWave e
                           como ferramenta de linha de comando)
  └── vendor
    └── ...
  └── inst_decode.sh    (script para traduzir de forma rápida uma
                           instrução em formato hexadecimal
                           usando o riscv-disassembler)
  └── LICENSE           (licença do sistema implementado)
  └── make.sh            (script para síntese e simulação de todas
                           as variantes do processador)
  └── gtkwave.sh         (script para invocar o GTKWave usando a
                           pasta do projeto como diretório raíz;
                           necessita do GTKWave instalado)
  └── rars.sh             (script para invocar o RARS presente em
                           tools/rars usando a pasta do projeto
                           como diretório raiz; necessita do
                           Java instalado)
  └── README.md          (README sobre o que é o projeto e como
                           utilizá-lo)

```

O trabalho também é organizado de forma a facilitar a migração para placas de desenvolvimento diferentes da *DE1-SoC* ou trocar o *soft-core* desenvolvido por outra implementação, independente da sua *ISA*. O *soft-core* implementado se encontra no caminho *core/risc\_v*. Assim, os demais módulos presentes na pasta *core* não dependem da arquitetura do processador, exceto a tela de *debug* presente na interface de vídeo, que mostra os nomes dos registradores segundo as

convenções da *RISC-V*. No entanto, a tela de *debug* foi projetada de modo a ser relativamente fácil customizá-la para utilização em outra arquitetura.

A Figura 3.1 mostra um diagrama de blocos simplificado da hierarquia dos módulos que compõe o projeto sintetizável. A principal estrutura, a `fpga_top` representa a placa de desenvolvimento, enquanto a `cpu` é o *soft-core* implementado.

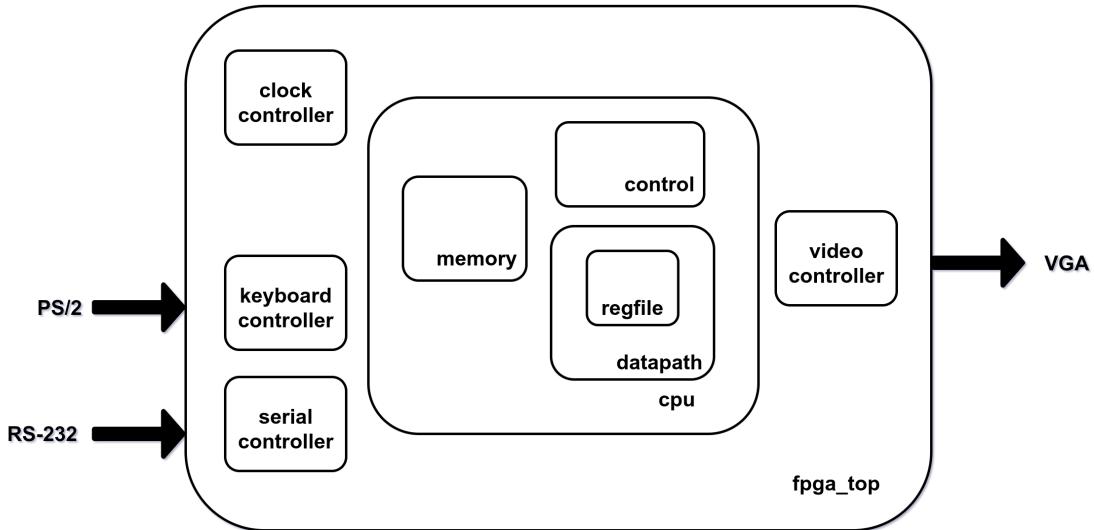


Figura 3.1: Diagrama de blocos simplificado do sistema

O arquivo `core/config.v` possui todas as opções de configuração, definição de parâmetros e endereçamento de memória dos módulos, facilitando escolher as extensões, microarquitetura e periféricos sintetizados. Outras pastas e arquivos relevantes serão discutidos nas próximas seções.

## 3.2 Implementação dos *soft-cores*

Todos os *soft-cores* implementados possuem execução em ordem, sem *branch prediction*, sem *caching* de memória e sem *Return Address Stack*. O processador é escalar e possui um único *hart*. Como a implementação atual só utiliza blocos de memória presentes no chip da FPGA, sem utilizar as memórias *SRAM* e *DRAM* externas presentes na placa de desenvolvimento, e também não faz uso de memória secundária, as operações de *load* e *store* transferem dados diretamente entre os registradores e os blocos de memória *M10K*. Os diagramas das microarquiteturas apresentados a seguir podem ser vistos em maior resolução no Anexo II.

### 3.2.1 Microarquitetura Uniciclo

Os processadores uniciclo com extensões I e IM são implementados conforme o diagrama da Figura 3.2. O módulo de controle é implementado somente com lógica combinacional, e a frequência máxima de operação é limitada pela instrução mais lenta do processador.

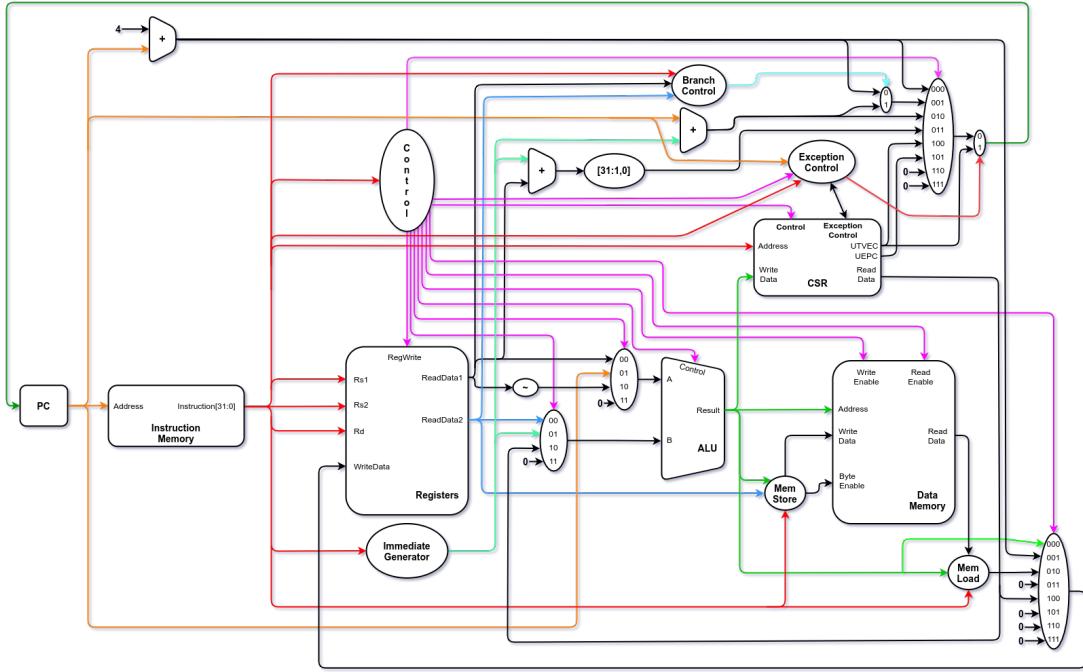


Figura 3.2: Diagrama da implementação das *ISAs* *RV32I* e *RV32IM* na microarquitetura uniciclo

A instrução recuperada da memória de instruções `core/memory/CodeMemory_Interface` no endereço apontado pelo registrador PC é lida pela unidade de controle `core/risc_v/Control_UNI.v`, que gera todos os sinais de controle enviados ao *datapath* `core/risc_v/Datapath_Uni.v`.

O banco de registradores `core/risc_v/Registers.v` também recebe alguns *bits* da instrução (para endereçamento dos registradores) e os dados para possivelmente serem escritos no registrador `rd` provenientes do multiplexador de *write-back*.

A unidade de lógica e aritmética `core/risc_v/ALU.v` possui um multiplexador em cada uma de suas entradas para selecionar os operandos.

Já a memória de dados `core/memory/DataMemory_Interface.v` recebe o endereço de escrita/-leitura vindo da saída da ULA, e suas operações de *load/store* são comandadas pelos controladores `core/memory/MemLoad.v` e `core/memory/MemStore.v`.

O banco de registradores de controle e estado `core/risc_v/CSRegisters.v` também é endereçado pela instrução, e o dado que possivelmente será gravado origina da ULA.

Há ainda as unidades de controle de exceção `core/risc_v/ExceptionControl.v`, responsável por redirecionar o contador de programa PC para o endereço do *kernel* quando ocorre uma exceção, como uma instrução inválida ou um endereço desalinhado. A unidade de controle de *branches* `core/risc_v/BranchControl.v` é responsável por rotear o endereço da próxima instrução caso ocorra um *branch* ou *jump*.

A única diferença entre a implementação da *ISA RV32I* e da *RV32IM* é a adição de novas operações na unidade de lógica e aritmética.

O processador uniciclo com extensão IMF é implementado conforme o diagrama da Figura 3.3. A unidade lógica e aritmética de ponto flutuante utiliza uma frequência de *clock* maior que a do resto do processador, e é o único módulo da implementação uniciclo que utiliza mais de um ciclo de relógio para realizar sua operação. A frequência máxima de operação do *clock* principal do processador continua limitada pela operação mais lenta.

Exceto o roteamento extra do banco de registradores `core/risc_v/FRegisters.v` de ponto flutuante e da ULA de ponto flutuante de precisão simples `core/risc_v/FPALU/FPALU.v`, o restante do circuito segue as mesmas conexões.

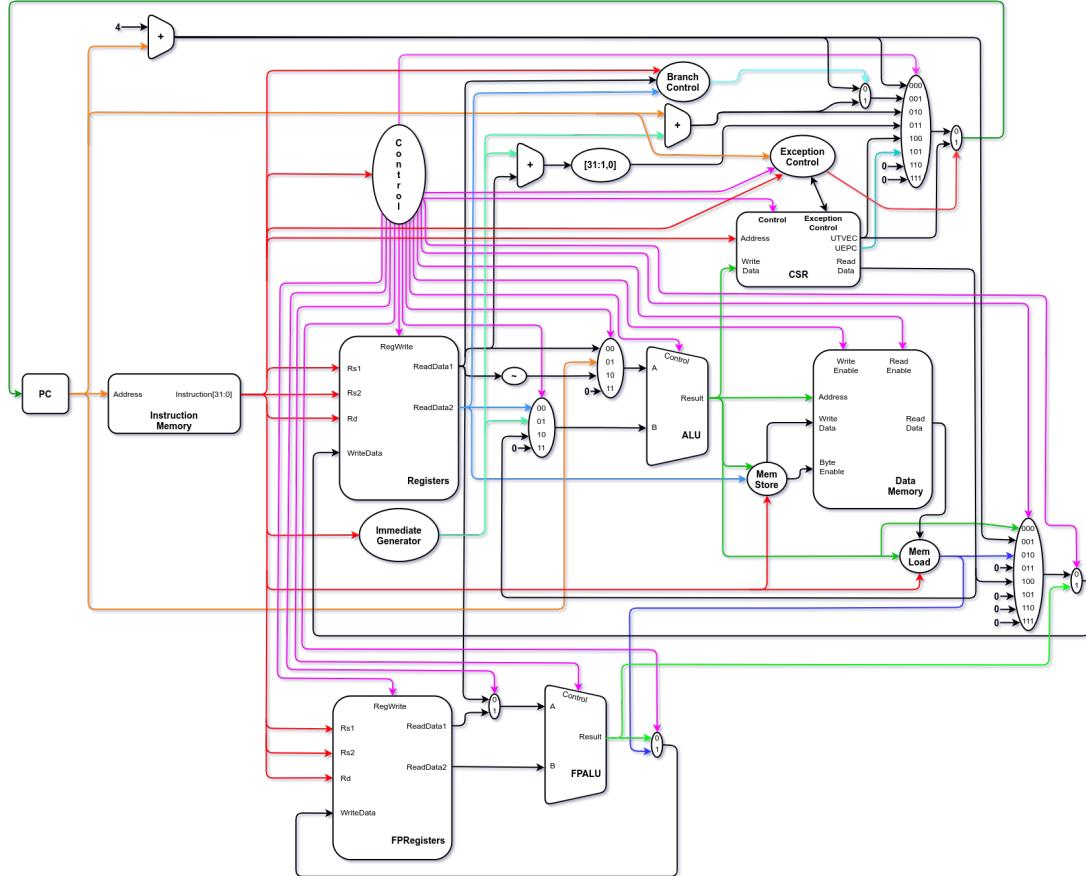


Figura 3.3: Diagrama da implementação da *ISA RV32IMF* na microarquitetura uniciclo

### 3.2.2 Microarquitetura Multiciclo

Os processadores multiciclo com extensões I e IM são implementados conforme o diagrama da Figura 3.4. A unidade de controle é implementada utilizando microcódigo para executar as instruções. Com isso, a frequência de operação do processador depende da operação mais lenta do microcódigo, e não da execução da instrução completa.

Enquanto as outras implementações usam arquitetura de memória *Harvard*, onde a memória de dados e a de instruções são separadas [48], o multiciclo utiliza arquitetura *von Neumann*, onde não há separação entre a memória de dados e a de texto [49].

O módulo de memória `core/memory/Memory_Interface.v` tem seu acesso controlado pelos blocos `core/memory/MemLoad.v` e `core/memory/MemStore`, além dos sinais vindos da unidade de controle `core/risc_v/Control_Multi.v`.

A outra grande diferença em relação à implementação do uniciclo é que a saída do bloco de memória, dos dados do banco de registradores, da saída da ULA e do banco de registradores de controle e estado. Há também o registrador `PCBack` que salva o contador de programa atual.

Além de alguns sinais em posições diferentes nos multiplexadores de seleção das entradas da ULA, de novo contador de programa e de *write back*, a atualização do contador de programa depende de uma comparação lógica.

Com o controlador `core/risc_v/Control_Multi.v` implementado por microcódigo, as instruções possuem variação na quantidade de ciclos para serem completadas.

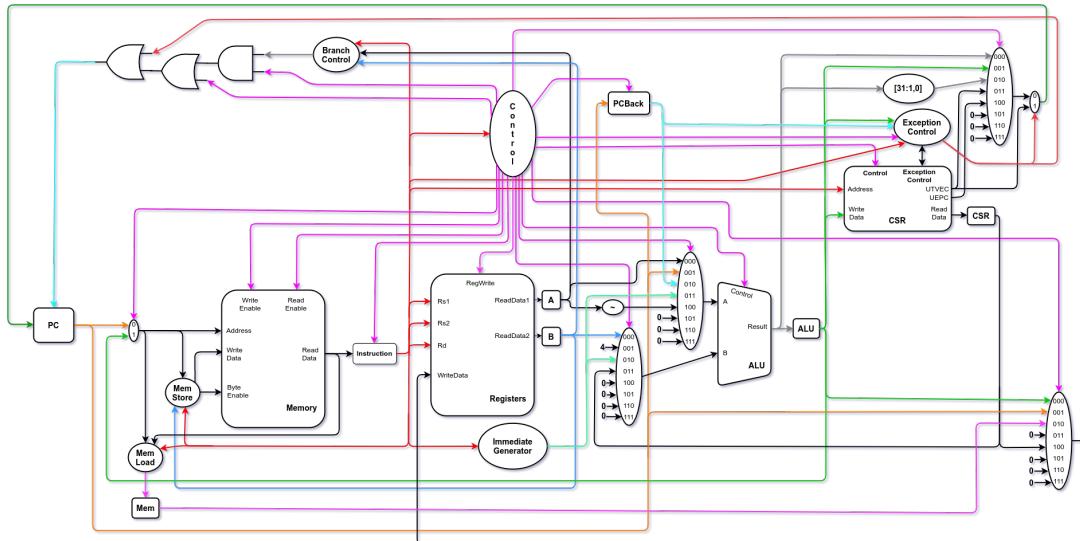


Figura 3.4: Diagrama da implementação das *ISAs* RV32I e RV32IM na microarquitetura multiciclo

O processador multiciclo com extensões IMF é implementado conforme o diagrama da Figura 3.5. A unidade lógica e aritmética de ponto flutuante utiliza uma frequência de *clock* mais alta que a do resto do processador, e possui um sinal de *ready* que causa o *stall* do clock principal do processador enquanto a operação de ponto flutuante não completa. Assim, a frequência do *clock* do processador é variável, já que em operações de ponto flutuante o ciclo do relógio é mais longo que em outras operações.

A diferença entre o conjunto de instruções com e sem a extensão **F** é a presença do banco de registradores e a unidade de ponto flutuante, que também possuem suas saídas registradas.

A presença dos registradores nas saídas dos módulos de onde dados são lidos servem para criar um atraso de um ciclo de *clock* nesses valores. Como a unidade de controle utiliza microcódigo, o dado lido no ciclo atual só será utilizado no próximo ciclo.

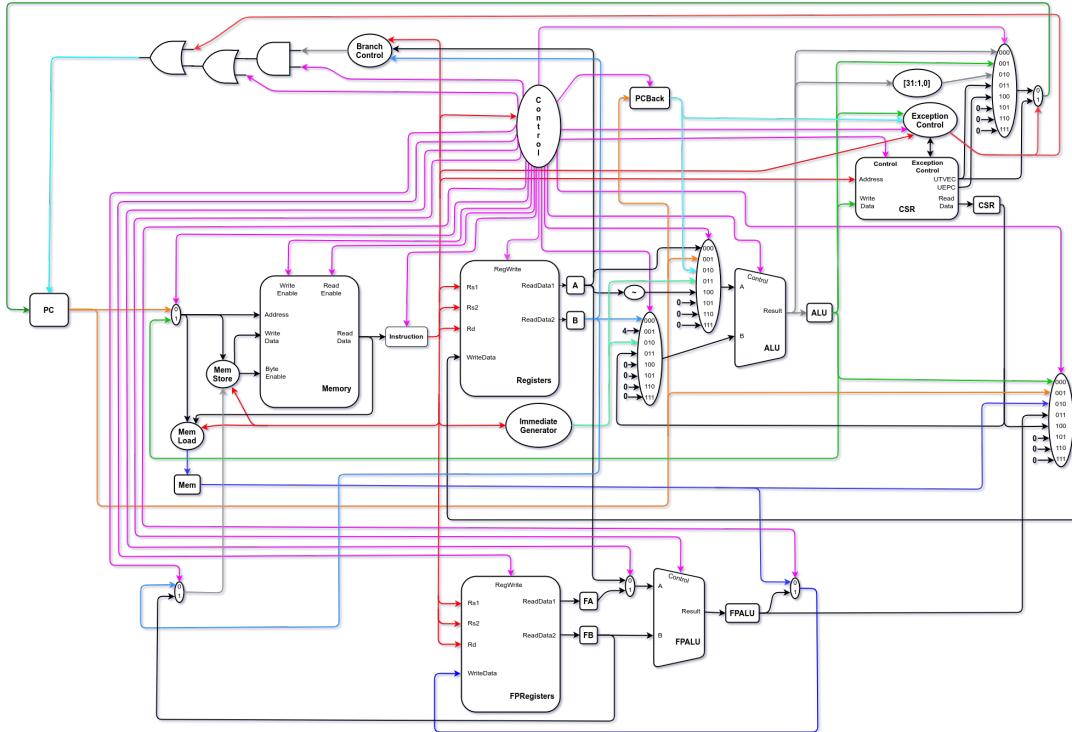


Figura 3.5: Diagrama da implementação da *ISA RV32IMF* na microarquitetura multiciclo

### 3.2.3 Microarquitetura *Pipeline* de 5 Estágios

Os processadores *pipeline* com extensões I e IM são implementados conforme o diagrama da Figura 3.6. Seus cinco estágios são:

1. *IF - Instruction Fetch*
2. *ID - Instruction Decode*
3. *EX - Execution*
4. *MEM - Memory Stage*
5. *WB - Write Back*

No primeiro estágio (*IF*) estão presentes o contador de programa, a memória de instruções e um somador que incrementa o PC em 4 bytes, que será o endereço da instrução seguinte caso não ocorra um *branch*, *jump*, exceção ou *syscall*.

Entre os estágios de *Instruction Fetch* e *Instruction Decode*, existe um grupo de registradores (*IFID*) responsável por passar os valores do primeiro estágio para o seguinte.

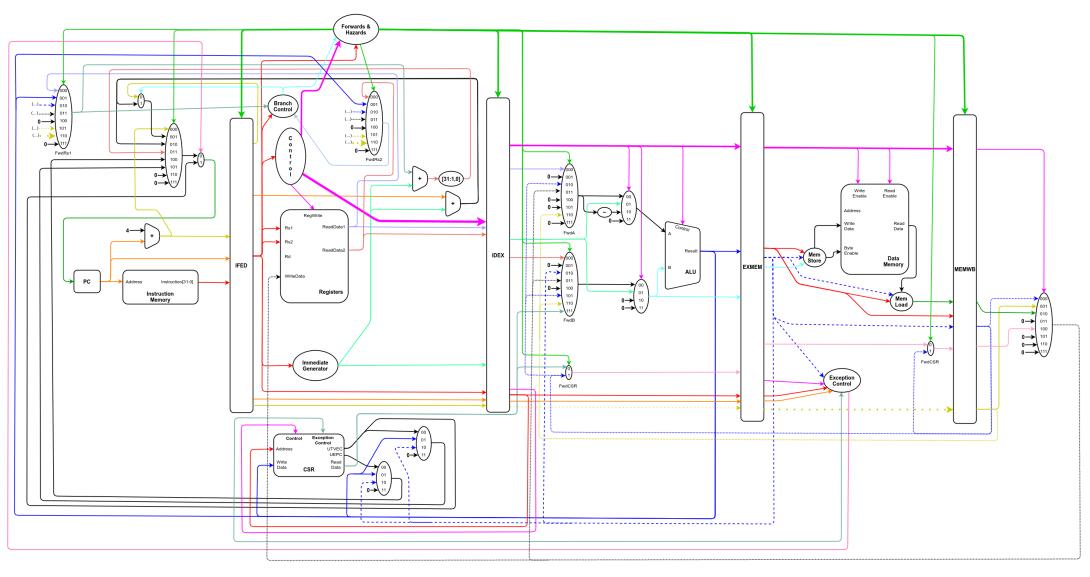


Figura 3.6: Diagrama das *ISAs* RV32I e RV32IM na microarquitetura *pipeline* de 5 estágios

No segundo estágio (*ID*) ocorre a decodificação da instrução. Também é o estágio onde ficam localizados o banco de registradores, os registradores de controle e estado, a unidade de geração de imediatos, a de controle de *branches* e *jumps*, além da unidade de *forwards* e *hazards* `core/risc_v/FwdHazardUnitM.v`.

A unidade de *forwards* e *hazards* é essencial para o bom funcionamento e desempenho do *pipeline*. Brevemente discutida na Seção 2.4.3, *hazards* ocorrem quando uma instrução a partir do segundo estágio tem como entrada um dado que ainda não foi calculado. É o caso do código a seguir:

---

```
0x00400000:      la  t1, 8
0x00400004:      lw  t0, 0(t1)
```

---

Quando a instrução `lw t0, 0(t1)` está calculando o endereço do *load* ( $t1 + 0$ ), o valor de `t1` ainda não foi salvo no banco de registradores, e por isso não pode ser lido. Os *forwards* identificam situações como essa e injetam o sinal calculado, “furando” a ordem do *pipeline*. Já em casos onde não é possível recuperar o valor necessário de uma etapa seguinte, ocorreu um *hazard* e um `nop` precisa ser inserido no *pipeline*, passando um ou mais ciclos sem processar novas instruções.

Entre os estágios de *Instruction Decode* e *Execution*, existe um grupo de registradores (*IDEX*) responsável por passar os valores do segundo estágio para o terceiro. No terceiro estágio (*EX*) ocorrem as operações lógicas e aritméticas. Os seletores das entradas da ULA recebem diversos sinais para possibilitar os *forwards* de dados.

A frequência máxima do *clock* do processador é limitada pela operação mais lenta da unidade lógica e aritmética.

Entre os estágios de *Execution* e *Memory Stage*, existe um grupo de registradores (*EXMEM*) responsável por passar os valores do terceiro estágio para o quarto. No quarto estágio (*MEM*) é

feito o acesso à memória. A unidade de controle de exceções também se encontra nesse estágio. A leitura da memória ocorre no ciclo positivo do relógio, enquanto a escrita é feita na borda de descida.

Entre os estágios de *Memory Stage* e *Write Back*, existe um grupo de registradores (*MEMWB*) responsável por passar os valores do quarto estágio para o quinto e último estágio. No quinto estágio (*WB*) é feita a escrita no banco de registradores. O sinal com o dado a ser escrito, bem como o endereço do registrador, é injetado no estágio *ID*, finalizando o ciclo de vida da instrução dentro do *pipeline*.

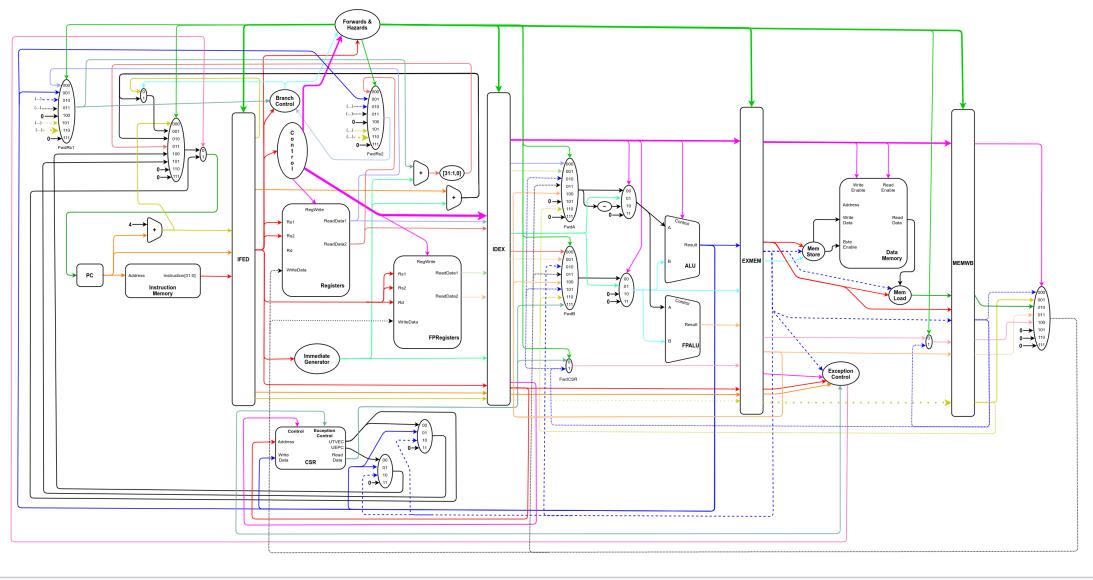


Figura 3.7: Diagrama da *ISA RV32IMF* na microarquitetura *pipeline* de 5 estágios

O processador *pipeline* com extensões **IMF** é implementado conforme o diagrama da Figura 3.7. Sua diferença entre o sistema descrito anteriormente é a presença do banco de registradores de ponto flutuante no estágio *ID* e da *FPALU* no estágio *EX*.

### 3.3 Chamadas de sistema

A pasta **system** contém a implementação das chamadas de sistema do processador. O código **assembly** deve incluir o arquivo **system/MACROS.s** no início do programa e o arquivo **system/SYSTEM.s** ao final do programa.

---

```
# Inicio do programa
.include "MACROS.s"

# Dados do programa
.data
...

```

```

# Instrucoes do programa
.text
...

# Chamadas de sistema
.include "SYSTEM.s"

```

---

O arquivo **MACROS.s** insere macros que testam se o programa está sendo executado no **Rars**, na **FPGA** ou no **ModelSim** para decidir o uso de determinadas *syscalls*, e também fornece a implementação por *software* de algumas instruções caso a extensão necessária não esteja implementada no processador, como instruções de multiplicação. Os endereços de memória dos periféricos acessados por *MMIO* também estão presentes como definições **.eqv** para facilitar a implementação dos programas.

Já o arquivo **SYSTEM.s** implementa o *kernel* do sistema, tratando exceções e executando as *syscalls*. As chamadas de sistema implementadas são apresentadas na Tabela 3.1.

Como o processador implementado não possui memória reservada para o *kernel*, sua posição inicial de memória começa imediatamente após a última instrução do programa implementado, e por isso deve ser incluído no final do arquivo. O arquivo de macros deve ser adicionado no início do programa, pois ele é responsável por gravar o endereço inicial do nível privilegiado no *CSR UTVEC* e ativar as interrupções.

Tabela 3.1: Tabela de *syscalls* implementadas.

<i>syscall</i>	a7	Argumentos	Operação
Print Integer	1 ou 101	a0 = inteiro a1 = coluna a2 = linha a3 = cores a4 = frame	Imprime no <i>frame</i> a4 o número inteiro a0 (complemento de 2) na posição (a1,a2) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .
Print Float	2 ou 102	fa0 = float a1 = coluna a2 = linha a3 = cores a4 = frame	Imprime no <i>frame</i> a4 o número de ponto flutuante fa0 na posição (a1,a2) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .
Print String	4 ou 104	a0 = endereço da string a1 = coluna a2 = linha a3 = cores a4 = frame	Imprime no <i>frame</i> a4 a <i>string</i> iniciada no endereço a0 e terminada em <i>NULL</i> na posição (a1,a2) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .

<i>syscall</i>	a7	Argumentos	Operação
Read Int	5 ou 105		Retorna em a0 o valor do inteiro em complemento de 2 lido do teclado.
Read Float	6 ou 106		Retorna em a0 o valor do <i>float</i> com precisão simples lido do teclado.
Read String	8 ou 108	a0 = endereço do buffer a1 = número máximo de caracteres	Escreve no <i>buffer</i> iniciado em a0 os caracteres lidos, terminando com um caracter <i>NULL</i> .
Exit	10 ou 110		Retorna ao sistema operacional. Na <i>DE1-SoC</i> , trava o processador.
Print Char	11 ou 111	a0 = char ASCII a1 = coluna a2 = linha a3 = cores a4 = frame	Imprime no <i>frame</i> a4 o carácter a0 na posição (a1,a2) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .
Read Char	12 ou 112		Retorna em a0 o valor ASCII do carácter lido do teclado.
Time	30 ou 130		Retorna o tempo do sistema em <i>ms</i> , com os 32 bits menos significativos em a0 e os 32 bits mais significativos em a1.
MIDI Out Assíncrono	31 ou 131	a0 = pitch a1 = duração (ms) a2 = instrumento a3 = volume	Gera o som definido e retorna imediatamente.
Sleep	32 ou 132	a0 = duração (ms)	Coloca o processador em <i>sleep</i> por a1 <i>ms</i> .
MIDI Out Síncrono	33 ou 133	a0 = pitch a1 = duração (ms) a2 = instrumento a3 = volume	Gera o som definido e retorna após o término da execução da nota.

<i>syscall</i>	a7	Argumentos	Operação
Print Integer	34 ou 134	a0 = inteiro a1 = coluna a2 = linha a3 = cores a4 = frame	Imprime no <i>frame</i> a4 o número inteiro a0 em formato hexadecimal na posição (a1,a2) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .
Print Integer Unsigned	36 ou 136	a0 = inteiro a1 = coluna a2 = linha a3 = cores a4 = frame	Imprime no <i>frame</i> a4 o número inteiro a0 sem sinal na posição (a1,a2) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .
Rand	41 ou 141		Retorna um número pseudorandômico de 32 bits em a0.
Draw Line	47 ou 147	a0 = $x_0$ a1 = $y_0$ a2 = $x_1$ a3 = $y_1$ a4 = cor a5 = frame	Desenha no <i>frame</i> a5 uma linha reta do ponto (a0,a1) até o ponto (a2,a3) com as cores a3[7:0] de <i>foreground</i> e a3[15:8] de <i>background</i> .
Read Char	48 ou 148	a0 = cor a1 = frame	Preenche o <i>frame</i> a1 com a cor a0.

As *ecalls* 1XX são utilizadas no *Rars* pelas ferramentas *Bitmap Display Tool* e *Keyboard Display MMIO Tool*, que foram customizadas para funcionar de maneira idêntica quando o programa é executado na *FPGA*.

### 3.4 Utilizando o RARS

Brevemente apresentado na Seção 2.3, a *IDE RARS* é uma ferramenta essencial da *toolchain* desse trabalho. Presente na pasta `tools/rars`, há duas versões customizadas do programa. As customizações permitem simular a saída de vídeo (exceto o menu *OSD*) e entrada do teclado da mesma forma que na *FPGA*.

Além da sua interface de edição de texto mostrada na Figura 2.10, há também a visualização do simulador. Nela é mostrado o código montado, os bancos de registradores e outras ferramentas como o simulador de *display*, como visto na Figura 3.8. É possível executar o código instrução por instrução, fazer o “*rewind*” da execução e definir *breakpoints*. Suas funções de *debug* são importantes tanto para validar o código produzido quanto para executar passo-a-passo junto com a *FPGA* para identificar discrepâncias na implementação do *hardware*.

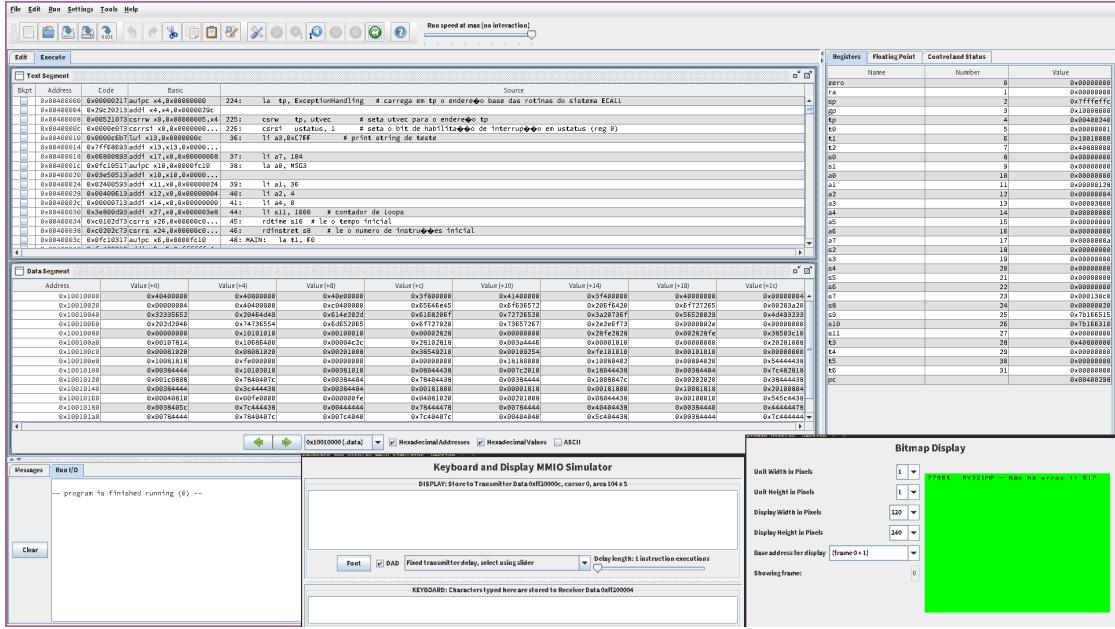


Figura 3.8: Exibição do *frame* de vídeo da *FPGA*

O *RARS* também é usado para gerar os arquivos *.mif* para inicializar a memória da *FPGA* clicando no menu **File > Dump Memory** ou usando o atalho de teclado **Ctrl+d**. São gerados dois arquivos, um contendo a memória de dados e o outro a memória de texto.

Os arquivos *core/default\_data.mif* e *core/default\_text.mif* são os arquivos gravados na memória da *FPGA* quando é realizada a síntese do processador. Esses arquivos também podem ser alterados a tempo de execução usando a ferramenta **Tools > In-System Memory Content Editor** do *Quartus*. A pasta *test/assembly\_testbenches* possui alguns programas em *assembly* para testar a *FPGA*. Já na pasta *test/mif\_library*, existem testes já montados prontos para gravação na placa de desenvolvimento.

### 3.5 Interface de vídeo e depuração

A interface de vídeo possui resolução de 320x240 *pixels* com 8 *bits* de cor para cada pixel. Efectivamente, a interface de vídeo possui 255 cores diferentes e uma cor utilizada como transparência, o magenta 0xC7. Ela também conta com dois *framebuffers*, permitindo renderizar duas imagens diferentes e alternar entre elas, ou se aplicado em um jogo, permite a transição de *frames* sem *flickering*: enquanto um *frame* é exibido, o outro *framebuffer* é construído com as imagens do próximo *frame*, e quando pronto, a tela é atualizada com o novo *frame* completamente renderizado.

A conexão do vídeo do sistema é feita por interface VGA, podendo se conectar a qualquer monitor com entrada VGA. A resolução real da interface é de 640x480 *pixels* com taxa de atualização de 59 Hz por questões de compatibilidade com os monitores. Cada *pixel* da interface de vídeo representa uma célula de 4 *pixels* na saída de vídeo real. A saída de vídeo VGA também possui 24 *bits* de cor, pois o controlador faz a conversão das cores em 8 *bits* para três canais de 8 *bits*,

um verde, um vermelho e um azul.



Figura 3.9: Exibição do *frame* de vídeo da *FPGA*

Acionando um *switch* da *FPGA*, é mostrado por cima do *frame* um *menu On Screen Display* que mostra o valor atual contido nos bancos de registradores do processador, incluindo os *CSRs* e, caso a extensão F esteja implementada, outro *switch* permite alternar entre a visualização dos registradores de ponto flutuante e os de ponto fixo.

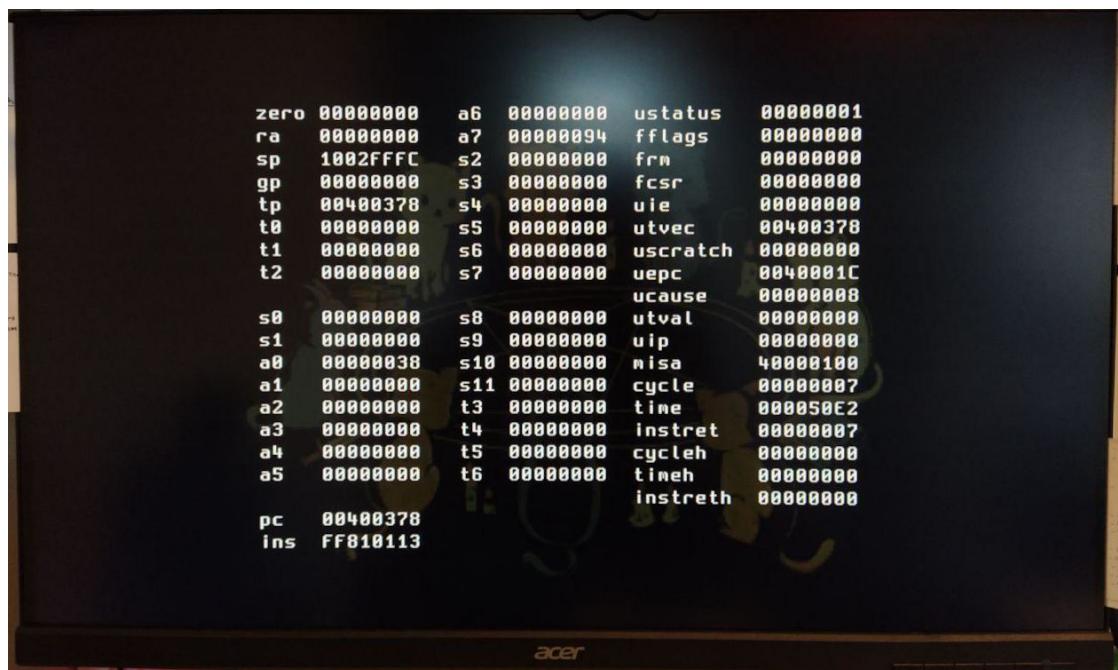


Figura 3.10: *Menu OSD* exibindo os valores dos registradores do processador

O menu *OSD* é implementado como uma matriz de 52x24 caracteres monoespaçados. Na matriz, os caracteres que não mudam com o tempo, como é o caso do nome dos registradores, são representados por um parâmetro correspondente ao próprio caracter. Já os valores que se alteram, como o valor dos registradores, são representados por um parâmetro *placeholder*, e o valor a ser mostrado na tela é obtido usando uma tabela de *lookup*. O projeto do menu *OSD* foi pensado de forma que possa ser modificado para expansão ou utilização em outras arquiteturas de processadores de maneira simples.

### 3.6 Configuração e síntese do processador pelo Quartus

O software utilizado para a síntese do processador, fornecimento de *IPs* como as de memória e operações de ponto flutuante, gravação do *soft-core*, dentre outras funcionalidades é o *Intel Quartus Lite v18.1*. Versões superiores são compatíveis com menos sistemas operacionais e/ou não possuem todos os *IPs* necessários para a síntese do processador.

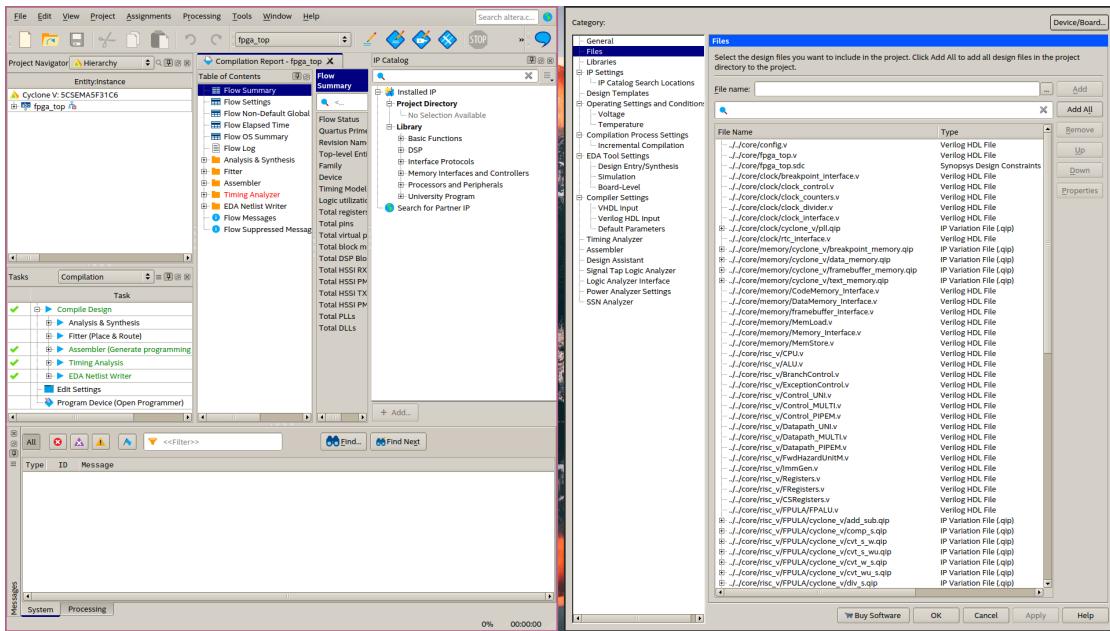


Figura 3.11: *Intel Quartus Lite v18.1* com a janela de configurações do projeto

Todas as configurações do projeto também podem ser alteradas manualmente no arquivo `project/de1_soc/fpga_top.qsf`. Para ativar ou desativar os pinos do *chip* da *FPGA* que conectam os periféricos da placa, é preferível que a edição seja feita diretamente no arquivo de configuração, comentando ou descomentando a declaração dos pinos.

Para realizar a síntese completa do processador para utilizá-lo na *FPGA*, basta acessar o menu **Processing > Start Compilation** ou utilizar o atalho **Ctrl + L** ou clicar no ícone de "Play" na barra de tarefas do programa. Assim, as etapas de Análise e Síntese, *Placing* e *Routing*, *Assembler* e *Timing Analysis* serão realizadas, e, caso não ocorram erros durante o processo, o *soft-core* estará pronto para ser gravado na *FPGA* utilizando o arquivo *.mif* gerado.

### 3.7 Simulação do processador pelo Quartus e ModelSim

O projeto possui um *testbench* em *Verilog* para simular as entradas e saídas da *FPGA* que o usuário operaria, como os botões e *switches*. Ele configura a rotina de reinicialização da placa após o *power-up* e define por quanto tempo a simulação será executada.

O script `test/simulation_scripts/de1_soc_rtl.do` é necessário para realizar a simulação de forma correta. O script gerado automaticamente pelo *Quartus* apresenta problemas que impedem que a simulação seja executada corretamente, além de não gerar o arquivo de saída da simulação no formato desejado.

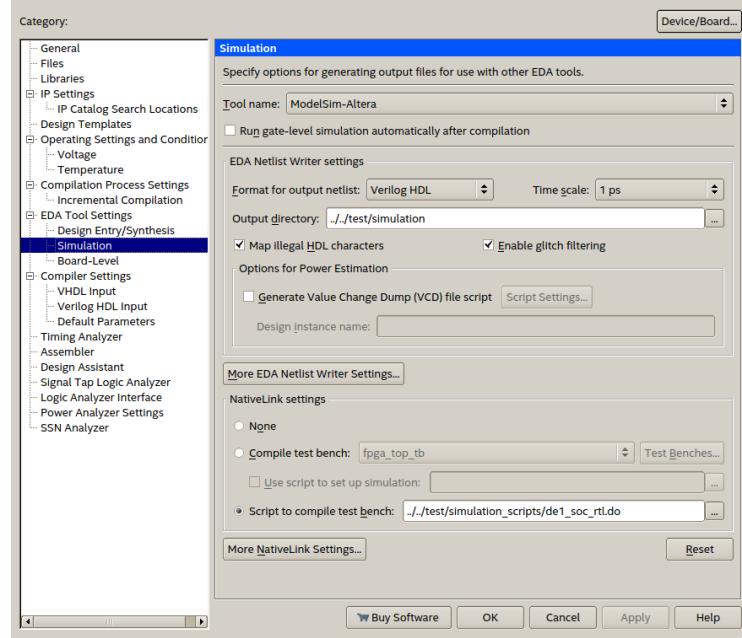


Figura 3.12: Janela de configuração da simulação no *Quartus*.

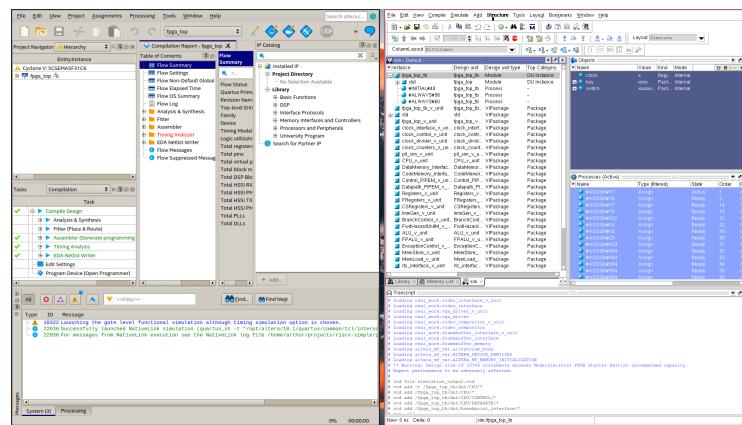


Figura 3.13: O script *NativeLink* invoca o *ModelSim* passando o script `.do` com as informações de como simular o sistema

Por limitações do *Quartus*, não é possível simular *FPGAs Cyclone V* a nível de portas lógicas,

somente sendo possível fazer a simulação *RTL*. Por outras limitações no programa, o *script .do* produzido manualmente só é executado usando o *menu Tools > Run Simulation Tool > Gate Level Simulation*, que, apesar do nome, executará uma simulação *RTL*. A opção *Tools > Run Simulation Tool > RTL Simulation* utiliza o *script .do* gerado automaticamente, e falha ao ser processado.

Ao executar a simulação, os *scripts NativeLink* iniciarão o programa *ModelSim*, carregando o *script .do* customizado, conforme mostrado na Figura 3.13. Ao finalizar a simulação, um arquivo *.vcd* será gerado e poderá ser analisado em *softwares* de visualização de formatos de onda, como o *GTKWave* ou o próprio *ModelSim*.

Ao carregar um arquivo *.vcd* no *GTKWave*, a hierarquia dos módulos é mostrada em uma árvore no canto superior esquerdo da tela. Ao clicar no nó desejado da árvore, os sinais do módulo serão mostrados no canto inferior esquerdo da tela. Para visualizá-lo, basta clicar e arrastar o sinal para a área *Signals*. A Figura 3.14 mostra uma visualização das formas de onda escolhidas. Na pasta *test/gtkwave* do projeto, existe um arquivo *.gtkw* para cada uma das nove configurações do *soft-core* com sinais predefinidos.

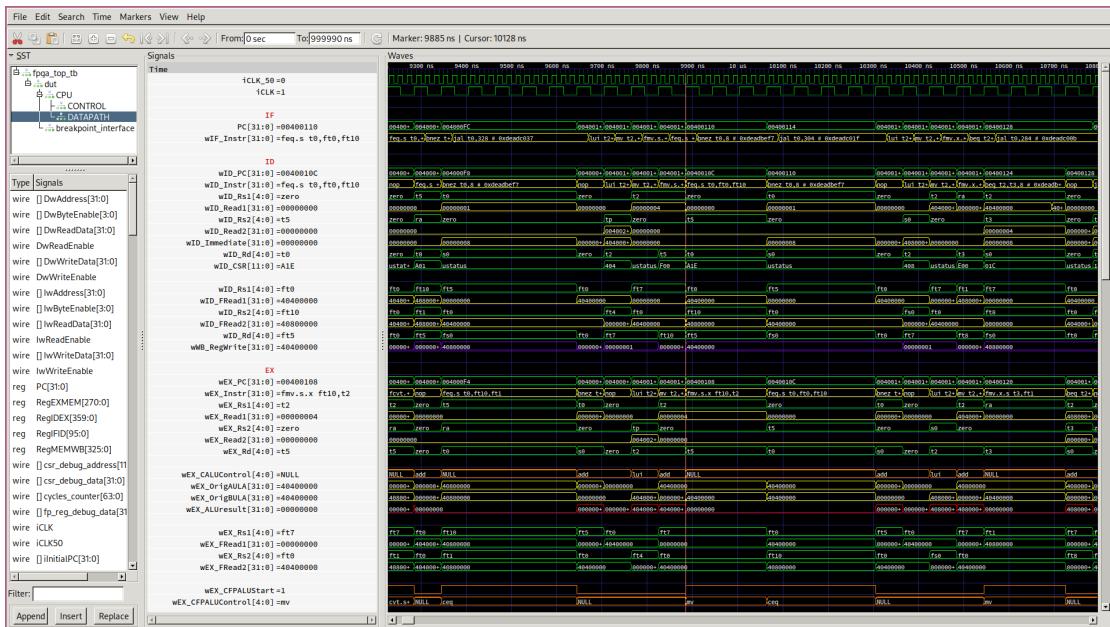


Figura 3.14: Visualização das formas de onda no *GTKWave*

É possível escolher as cores dos sinais para melhor destaque, bem como utilizar arquivos e programas para a tradução de valores dos sinais. Na pasta *test/gtkwave/translation\_files* se encontram arquivos *.txt* para tradução dos códigos hexadecimais dos seletores de registradores e de controle das unidades de lógica e aritmética. Assim, a visualização da forma de onda mostra os mnemônicos dos sinais, facilitando sua compreensão. O programa *riscv-decode* presente em *tools/riscv-disassembler/build* desmonta as instruções e as exibe na visualização.

### 3.8 Script `make.sh`

A pasta `test/sof_library` contém os arquivos `.sof` com as nove variações da última versão do processador, prontos para serem gravados na *FPGA*. Para facilitar a geração das nove variantes, o *bash script* `make.sh` foi criado para automatizar a síntese, salvando as novas versões na pasta `test/sof_library`. O *script* também permite realizar somente a etapa de análise das variantes para confirmar que alterações feitas no código não introduziram erros de compilação, uma vez que é um processo muito mais ágil que realizar a síntese completa.

Além disso, o *script* também possui opção para simular *RTL* todas as variantes do processador, salvando os *logs* e arquivos `.vcd` na pasta `test/simulation`. A pasta `test/simulation` é ignorada pelo *git*, pois os arquivos de forma de onda podem ficar grandes demais a ponto de inviabilizar seu versionamento.

### 3.9 Uso da *FPGA* DE1-SoC

A placa de desenvolvimento *terasIC DE1-SoC* utilizada no projeto é mostrada na Figura 3.15.

Os botões e *switches* mostrados na Figura 3.15 são utilizados para controlar as características do *clock* do processador, fazer seu *reset* e controlar o *menu OSD* de depuração. A função de cada *input* é:

- KEY0: Reset do processador;
- KEY1: Seletor de divisor de *clock* lento ou rápido;
- KEY2: Seletor de *clock* manual ou automático;
- KEY3: Gerador de *clock* manual;
- SW0: Bit [0] do divisor de *clock*;
- SW1: Bit [1] do divisor de *clock*;
- SW2: Bit [2] do divisor de *clock*;
- SW3: Bit [3] do divisor de *clock*;
- SW4: Bit [4] do divisor de *clock*;
- SW5: Temporizador para *stall* do processador;
- SW6: Seletor de *framebuffer* a ser exibido;
- SW7: Seletor de banco de registradores no *menu OSD*;
- SW8: Sem função;
- SW9: Habilita o *menu OSD*;

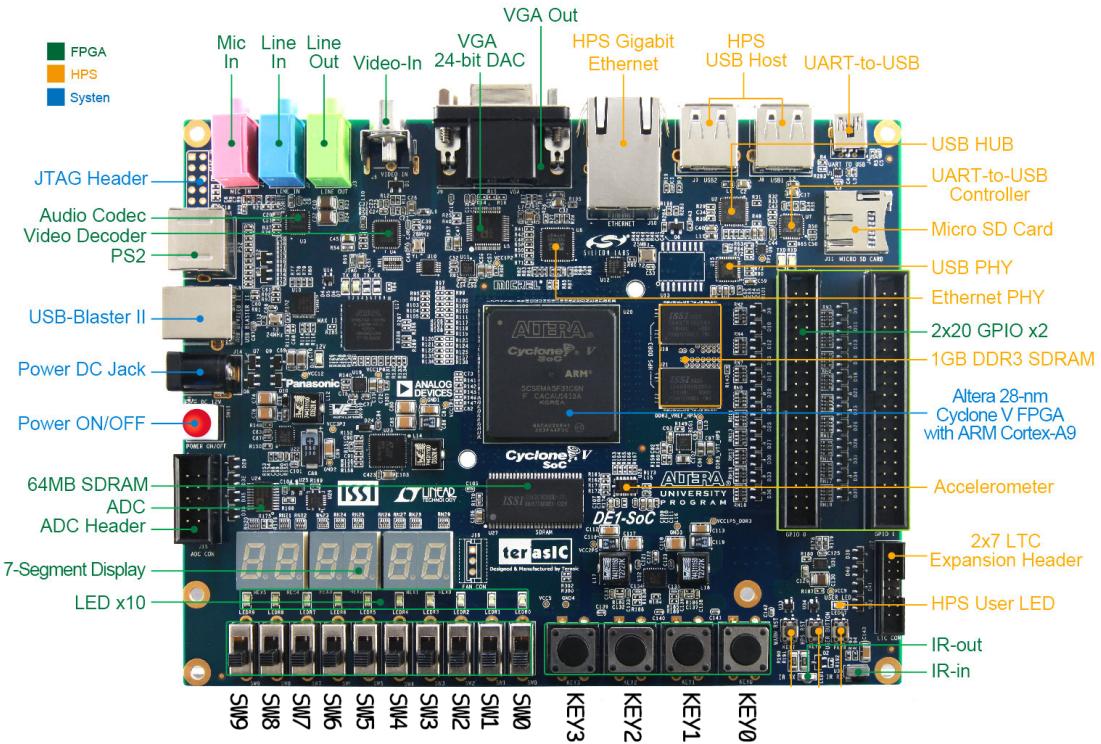


Figura 3.15: Placa de desenvolvimento *terasIC DE1-SoC*. Fonte: [50]

O procedimento recomendado para inicialização do processador após o *Programmer* do *Quartus* programar a *FPGA* com um novo arquivo *.mif* é: Pressionar e soltar a *KEY2* para ativar o *clock* automático; pressionar e soltar a *KEY0* para dar o *reset* dos estados do processador e, caso queira uma execução mais rápida, pressionar e soltar a *KEY1* para mudar para um divisor de *clock* mais veloz. A frequência máxima de operação do *clock* é de 50MHz, e ocorre na opção de divisor rápido com o divisor 5'b1.

Utilizando a saída de vídeo, o sistema pode executar programas gráficos como jogos, ou pode ser usado simplesmente como ferramenta de *debug*. Ativando o menu *OSD* e utilizando o *clock* manual, é possível ver a progressão dos registradores do processador instrução por instrução.

O processador também pode receber *inputs* do usuário utilizando um teclado *PS/2*. A leitura do teclado é realizada por meio de *polling* do endereço do *buffer* do teclado.

### 3.10 Observações Finais do Sistema Proposto

No presente capítulo vimos o detalhamento da implementação e das ferramentas utilizadas no ambiente de aprendizado, e como utilizá-las. No próximo capítulo, trataremos dos resultados do trabalho, apresentando tabelas comparativas entre as implementações, análise de performance e visualizações das formas de onda do sistema simulado.

# Capítulo 4

## Resultados

Após a exposição dos componentes do ambiente de aprendizado e estrutura das implementações do processador, é possível realizar análises quantitativas e qualitativas dos resultados obtidos.

As seções desse capítulo explorarão os resultados da síntese e simulação de cada versão, além de apresentar resultados de *benchmarks* sintéticos a fim de comparar o desempenho de cada implementação e sua viabilidade de uso.

O código-fonte e demais arquivos da plataforma *RISC-V SiMPLE* desenvolvida está disponível no link <https://github.com/LAICO-UnB/riscv-simple> com licença *open-source BSD 3-Clause*. O código-fonte e .pdf dessa monografia está disponível no link <https://github.com/arthurbeggs/monografia> com licença *open-source BSD 3-Clause*.

### 4.1 Síntese dos *soft-cores*

As nove diferentes implementações do processador foram geradas usando o *script* `./make.sh -simulate`, produzindo os arquivos .sof para gravação na *FPGA*, os .vcd de simulação em forma de onda e os .rpt de resumo do *Quartus*.

Os dados da Tabela 4.1 foram obtidos dos arquivos .rpt e executando os arquivos .sof na placa *DE1-SoC*. Cada *ISA* foi carregada com um *benchmark* específico para seu conjunto de instruções. Os códigos-fonte podem ser encontrados em `test/assembly_testbench` e suas versões montadas estão disponíveis na pasta `test/mif_library`.

Tabela 4.1: Características dos sistemas implementados

	ALMs	Regs	Pins	Mem Bits	DSPs	PLLs	Max Clk	
Uniciclo	Máximo	32070	XXXXXX	457	4065280	87	1	50MHz
	RV32I	4123	3160	103	2805792	0	1	12.5MHz
	RV32IM	7047	3179	103	2805792	12	1	12.5MHz
	RV32IMF	9411	5558	103	2853408	18	1	3.5Mhz

	ALMs	Regs	Pins	Mem Bits	DSPs	PLLs	Max Clk	
Máximo	32070	XXXXXX	457	4065280	87	1	50MHz	
Multiciclo	RV32I	4102	3444	103	2805792	0	1	25MHz
	RV32IM	6726	3471	103	2805792	12	1	25MHz
	RV32IMF	9108	5737	103	2853408	18	1	25MHz
Pipeline	RV32I	4605	4139	103	2805792	0	1	25MHz
	RV32IM	7376	4145	103	2805792	12	1	25MHz
	RV32IMF	9750	6568	103	2853408	18	1	25MHz*

Ao analisar a tabela, podemos tirar as seguintes conclusões:

- O número de *PLLs* e de pinos não muda entre as implementações, pois somente um *PLL* é utilizado para gerar os sinais de relógio da *FPGA*, e o *pinout* do módulo *top level* não é alterado entre as versões do processador. Variação nesses valores representaria um erro;
- O número de *DSPs* é 0 para a *ISA RV32I*, 12 para a *RV32IM* e 18 para a *RV32IMF*. Os *DSPs* apenas são utilizados nas operações de *mul/div* e ponto flutuante;
- A quantia de *bits* de memória utilizados permanece igual para todas as implementações das *ISAs RV32I* e *RV32IM*. Todas as implementações da *RV32IMF* também utilizam a mesma quantia de *bits*, que é levemente maior que nas outras duas arquiteturas;
- Como é de se esperar, a quantia de *ALMs* e *registradores* aumenta ao implementar mais extensões numa mesma microarquitetura;
- A microarquitetura multiciclo utiliza a menor quantia de recursos entre as três, resultado esperado já que sua implementação reutiliza estruturas como a ULA na sua execução por microcódigo, enquanto as outras arquiteturas utilizam mais de um somador com funções específicas em seu *datapath*;
- A frequência máxima de operação do multiciclo se manteve constante para as três *ISAs* implementadas e teve bom desempenho;
- A frequência de operação do uniciclo foi a mais baixa entre os sistemas implementados, como era esperado. Com o uso de operações de ponto flutuante, sua frequência máxima foi bastante penalizada;
- A implementação da *ISA RV32IMF* no *pipeline* apresenta erros devido a *forwards* e *hazards* não tratados ou tratados de maneira incorreta. Para as extensões *I* e *IM* sua frequência máxima é de 25MHz, como no multiciclo. A extensão *IMF* apresenta erros durante sua execução, então não é possível confirmar que sua frequência máxima também é de 25MHz.

## 4.2 Formas de Onda das Simulações

As Figuras 4.1, 4.2 e 4.3 mostram as visualizações criadas para as simulações dos *soft-cores*

*RV32IMF* uniciclo, multiciclo e *pipeline* respectivamente. Nelas, as instruções passam por um *desassembler*, os registradores são mostrados com seus nomes mnemônicos e os sinais possuem cores diferentes dependendo de sua origem.

Com essas visualizações, o processo de depuração do processador é facilitado. Alguns *bugs* do *pipeline* só puderam ser identificados graças à simulação. Os sinais adicionados são suficientes para a maioria das inspeções, mas se necessário, novos sinais podem ser adicionados.

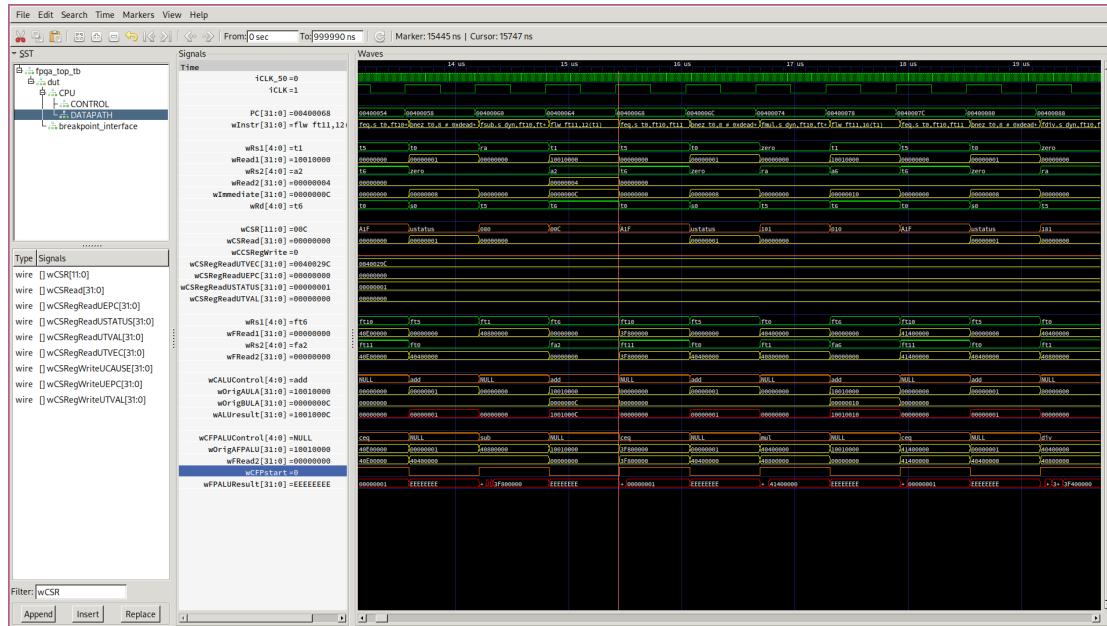


Figura 4.1: Visualização das formas de onda *soft-core RV32IMF* uniciclo

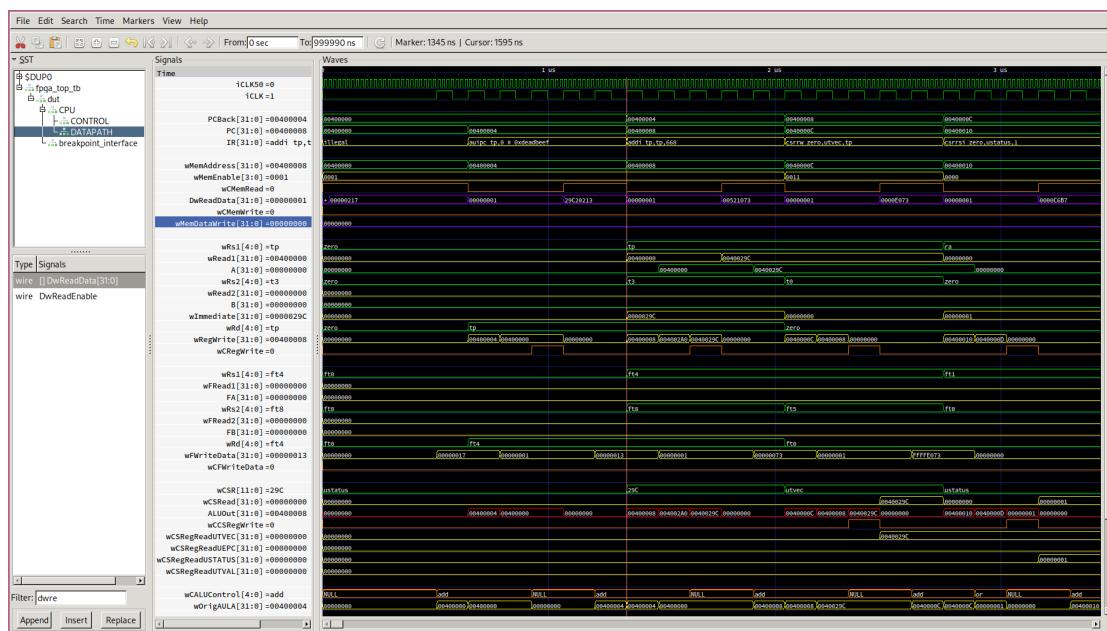


Figura 4.2: Visualização das formas de onda *soft-core RV32IMF* multiciclo

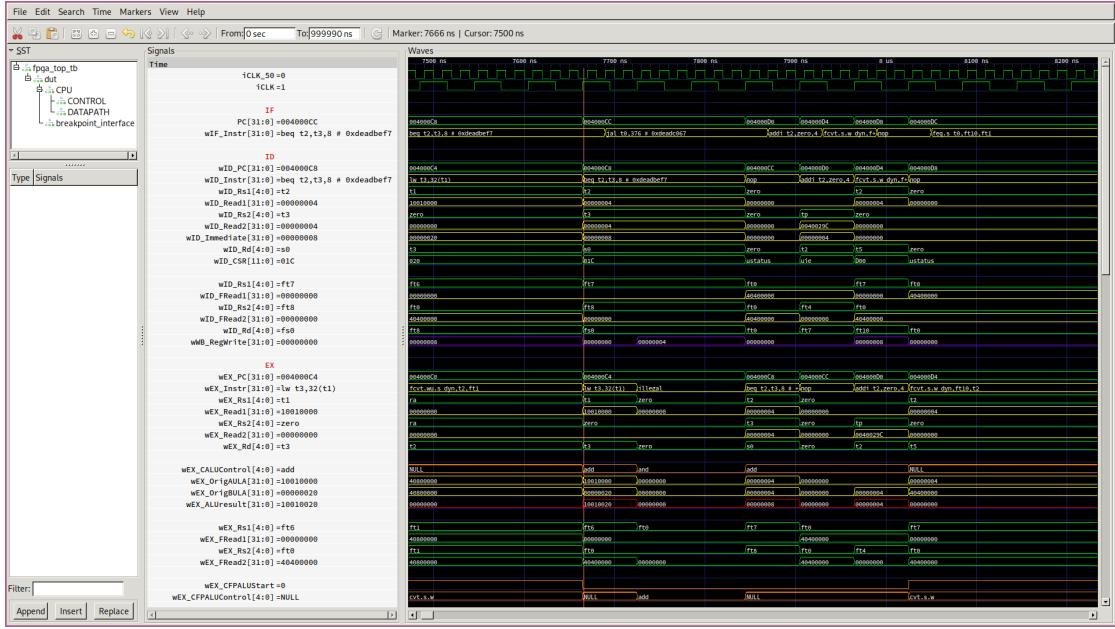


Figura 4.3: Visualização das formas de onda *soft-core RV32IMF pipeline*

### 4.3 Benchmarks

Para analisar o desempenho das diferentes implementações, três *benchmarks* foram realizados. Cada um dos *benchmarks* foi criado para rodar em uma *ISA* específica. Assim, temos um teste para a arquitetura *RV32I*, um segundo para a *RV32IM* e por último, um para a *RV32IMF*. Pela escassez de instruções implementadas por *software* caso uma extensão não esteja disponível, fazer um único arquivo de teste e executá-lo nas 9 implementações não é ideal.

Os três *benchmarks* realizam 1000 ciclos de operações específicas de sua extensão. O valor do registrador de estado *time* será usado para verificar quantos milissegundos foram necessários para completar o teste. Essa análise tem como objetivo comparar o desempenho das diferentes microarquiteturas para o mesmo *workload* dispondo do mesmo conjunto de instruções. Os testes são executados utilizando a frequência máxima da implementação, conforme a Tabela 4.1 e seus códigos-fonte se encontram em `test/assembly_testbench`.

Tabela 4.2: Comparativo de desempenho de cada *ISA* em microarquiteturas distintas

		RV32I	RV32IM	RV32IMF
Uniciclo	time	0x00000001E	0x000000012	0x000000034
	clock	12.5MHz	12.5MHz	3.5MHz
Multiciclo	time	0x000000048	0x00000002E	0x00000003D
	clock	25MHz	25MHz	25MHz
<i>Pipeline</i>	time	0x000000015	0x000000010	erro
	clock	25MHz	25MHz	25MHz

Pelos resultados obtidos, é possível ver que, apesar do multiciclo ter frequência mais alta que o uniciclo, em todas as implementações ele demorou mais a completar os testes. Para as *ISAs* *RV32I* e *RV32IM*, o *pipeline* foi a versão mais rápida. Na *ISA* *RV32IMF*, existem erros na execução do *pipeline*, e por isso não foi possível compará-la com as demais.

#### 4.4 Observações Finais dos Resultados

Vimos no capítulo o resultado da execução de pequenos testes da plataforma desenvolvida, tanto em simulação por forma de onda quanto na execução de *benchmarks* na *FPGA DE1-SoC*. Das nove implementações, uma apresenta falhas a tempo de execução, a *RV32IMF* em *pipeline* de cinco estágios. No entanto, a configuração feita para a ferramenta de simulação de forma de onda trará mais um instrumento para encontrar e solucionar o defeito de código.

O próximo capítulo encerrará o presente trabalho fazendo observações pertinentes aos resultados obtidos, à viabilidade do uso da plataforma desenvolvida para os propósitos desejados e perspectivas futuras, tratando de possíveis melhorias e expansão do escopo.

# Capítulo 5

## Conclusões

Nos capítulos anteriores foram explorados conceitos sobre organização e arquitetura de computadores, *overviews* de arquiteturas famosas e um aprofundamento na especificação do conjunto de instruções *RISC-V*. Além disso, foi descrito o que é, como funciona e quais são as peças do Ambiente de Aprendizado de microarquiteturas Uniciclo, Multiciclo e Pipeline, o *RISC-V SiMPLE*, desde o uso de ferramentas preexistentes até a criação de materiais próprios.

### 5.1 Objetivos Alcançados

O projeto de melhoria e documentação da plataforma utilizada no laboratório obteve resultados satisfatórios. Tanto a presença da documentação da arquitetura quanto as melhorias em código devem permitir que o uso da ferramenta seja facilitado.

A implementação do conjunto de instruções *RV32IMF* em *pipeline* de cinco estágios ainda apresenta alguns *bugs* de execução. O trabalho conseguiu corrigir alguns desses erros, mas a implementação ainda não se encontra no estado de ter a garantia de funcionamento correto do código ao executar operações de ponto flutuante sem a inserção de `nops` no código, o que é caracterizado como uma prática ruim.

Em relação às oito outras implementações, não foram encontrados erros nos testes realizados, e seu uso pode ser considerado seguro.

A expansão das ferramentas de auxílio ao desenvolvimento e depuração como a automação do processo de síntese de todas as versões do processador e o uso de ferramentas mais completas como o *GKWave* trouxeram maior observabilidade dos problemas, suas causas e caminhos para solucioná-los, acelerando o ciclo de desenvolvimento do projeto. Com isso, o resultado final se alinha à maioria das suas expectativas.

## 5.2 Perspectivas Futuras

Apesar do trabalho ter tido bom progresso, ainda existem diversas possíveis melhorias, como:

- Deixar a implementação do *pipeline bug-free*;
- Simplificar partes do projeto de *hardware* para melhorar o desempenho do sistema;
- Implementar uma versão de 64 *bits* do sistema;
- Implementar uma *ISA* diferente usando a plataforma como base, permitindo a escolha da arquitetura pelo arquivo `config.v`.

## 5.3 Palavras Finais

Com o recente sucesso dos processadores *ARM M1* lançados pela *Apple*, e com os processadores *ARM Graviton* disponíveis no serviço de servidores em nuvem da *Amazon*, é uma possibilidade forte que o desenvolvimento de plataformas *RISC-V* para uso geral desacelere e o mercado *ARM* cresça ainda mais. De um ponto de vista mercadológico, o desenvolvimento de uma plataforma de aprendizagem em *ARM* aumentaria as chances de aplicação direta do conteúdo aprendido no ambiente de trabalho.

Atualmente, algumas versões de seu conjunto de instruções foram abertas. Porém, a incerteza quanto a possíveis problemas de licenciamento para desenvolvimento e distribuição de uma solução *ARM* no início do projeto inviabilizaram seu uso para o presente trabalho.

Mesmo assim, a *RISC-V* já possui um bom alicerce, tem futuro promissor e a escolha da *ISA* para o desenvolvimento do trabalho se mostrou uma decisão acertada. Mercados emergentes também abrem diversas portas e se espera que o conhecimento adquirido desenvolvendo e utilizando a plataforma se provará de grande valia.

# REFERÊNCIAS BIBLIOGRÁFICAS

- [1] WATERMAN, A. et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 20191213*. 2019.
- [2] WATERMAN, A. et al. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 20190608*. 2019.
- [3] LYONS, J. *Natural language and universal grammar*. Cambridge England New York: Cambridge University Press, 1991. ISBN 9780521246965.
- [4] BLOCK diagram of a basic computer with uniprocessor CPU. (CC BY-SA 4.0). 2015. Disponível em: <<https://en.wikipedia.org/wiki/File:ABasicComputer.gif>>.
- [5] BLEM, E. et al. Isa wars: Understanding the relevance of isa being risc or cisc to performance, power, and energy on modern architectures. *ACM Trans. Comput. Syst.*, Association for Computing Machinery, 2015. ISSN 0734-2071. Disponível em: <<https://doi.org/10.1145/2699682>>.
- [6] MIPS Technologies, Inc. Enhances Architecture to Support Growing Need for IP Re-Use and Integration. Disponível em: <<https://web.archive.org/web/20181201180124/https://www.thefreelibrary.com/MIPS+Technologies%2c+Inc.+Enhances+Architecture+to+Support+Growing+Need...-a054531136>>.
- [7] PATTERSON, D. *Computer organization and design : the hardware/software interface*. [S.l.]: Morgan Kaufmann, 2014. ISBN 9780124077263.
- [8] SWEETMAN, D. *See MIPS run*. [S.l.]: Morgan Kaufmann Publishers, 1999. ISBN 1558604103.
- [9] MIPS architecture: Uses. Disponível em: <[https://en.wikipedia.org/wiki/MIPS\\_architecture#Uses](https://en.wikipedia.org/wiki/MIPS_architecture#Uses)>.
- [10] WAVE Computing and MIPS Wave Goodbye. Disponível em: <<https://semikiwi.com/ip/284876-wave-computing-and-mips-waves-goodbye>>.
- [11] SOME facts about the Acorn RISC Machine. Disponível em: <[https://groups.google.com/g/comp.arch/c/hPsDLEPf2eo/m/nvJR\\_d7nnnyYJ](https://groups.google.com/g/comp.arch/c/hPsDLEPf2eo/m/nvJR_d7nnnyYJ)>.
- [12] ARM'S Reach: 50 Billion Chip Milestone. Disponível em: <<https://web.archive.org/web/20160624003846/http://www.broadcom.com/blog/chip-design/arms-reach-50-billion-chip-milestone-video/>>.

- [13] NEW No1 Supercomputer: Fugaku in Japan, with A64FX, take Arm to the Top with 415 PetaFLOPs. Disponível em: <<https://www.anandtech.com/show/15869/new-1-supercomputer-fujitsus-fugaku-and-a64fx-take-arm-to-the-top-with-415-petaflops>>.
- [14] APPLE M1 Benchmarks Are Here – Apple Delivered Performance and Efficiency. Disponível em: <<https://web.archive.org/web/20201228121118/https://borderpolar.com/2020/11/21/apple-m1-benchmarks>>.
- [15] ARM Architecture Reference Manual - Armv8, for Armv8-A architecture profile. Disponível em: <<https://developer.arm.com/documentation/ddi0487/latest>>.
- [16] RARS. Disponível em: <<https://github.com/TheThirdOne/rars>>.
- [17] MARS (MIPS Assembler and Runtime Simulator). Disponível em: <<http://courses.missouristate.edu/KenVollmar/mars/index.htm>>.
- [18] A generic 4-stage pipeline (CC BY-SA 3.0). Disponível em: <[https://en.wikipedia.org/wiki/File:Pipeline,\\_4\\_stage.svg](https://en.wikipedia.org/wiki/File:Pipeline,_4_stage.svg)>.
- [19] COLLEGE, A.; DOYLE, C.; RYNNING, A. *FPGA Flexible Architecture - Olin College of Engineering*. Disponível em: <[http://ca.olin.edu/2005/fpga\\_dsp/images/fpga001.jpg](http://ca.olin.edu/2005/fpga_dsp/images/fpga001.jpg)>.
- [20] STANNERED. *Switch Box*. Disponível em: <[https://en.wikipedia.org/wiki/File:Switch\\_box.svg](https://en.wikipedia.org/wiki/File:Switch_box.svg)>.
- [21] INTEL. *Cyclone V SoC FPGA Architecture*. Disponível em: <<https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v.html>>.
- [22] INTEL. *ALM High-Level Block Diagram for Cyclone V Devices*. Disponível em: <[https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv\\_5v2.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v2.pdf)>.
- [23] DOCUMENTATION/PLATFORMS/RISCV. Disponível em: <<https://wiki.qemu.org/Documentation/Platforms/RISCV>>.
- [24] RISC-V BOOM - The Berkeley Out-of-Order RISC-V Processor. Disponível em: <<https://boom-core.org>>.
- [25] UC Berkeley Architecture Research. Disponível em: <<https://bar.eecs.berkeley.edu/>>.
- [26] BOOM Open Source RISC-V Core Runs on Amazon EC2 F1 Instances. Disponível em: <<https://www.cnx-software.com/2018/12/13/boom-risc-v-core-amazon-ec2-f1>>.
- [27] SHAKTI - Open Source Processor Development Ecosystem. Disponível em: <<https://shakti.org.in>>.
- [28] SHAKTI RISC-V based Processor: the First Open Source Indian Chip. Disponível em: <<https://www.open-electronics.org/shakti-risc-v-based-processor-the-first-open-source-indian-chip>>.

- [29] WESTERN Digital to Use RISC-V for Controllers, Processors, Purpose-Built Platforms. Disponível em: <<https://www.anandtech.com/show/12133/western-digital-to-develop-and-use-risc-v-for-controllers>>.
- [30] RISC-V in Nvidia. Disponível em: <<https://riscv.org/wp-content/uploads/2017/05/Tue1345pm-NVIDIA-Sijstermans.pdf>>.
- [31] NEW Part Day: A RISC-V CPU For Eight Dollars. Disponível em: <<https://hackaday.com/2019/02/14/new-part-day-a-risc-v-cpu-for-eight-dollars/>>.
- [32] HIFIVE1 Rev B. Disponível em: <<https://www.sifive.com/boards/hifive1-rev-b>>.
- [33] HIFIVE Unmatched. Disponível em: <<https://www.sifive.com/boards/hifive-unmatched>>.
- [34] BEAGLEV The First Affordable RISC-V Computer Designed to Run Linux. Disponível em: <<https://beaglev.seeed.cc/>>.
- [35] SIFIVE Tapes Out First 5nm TSMC 32-bit RISC-V Chip with 7.2 Gbps HBM3. Disponível em: <<https://www.tomshardware.com/news/openfive-tapes-out-5nm-risc-v-soc>>.
- [36] GCC RISC-V Options. Disponível em: <<https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html>>.
- [37] LLVM Clang RISC-V Now Supports LTO. Disponível em: <<https://riscv.org/news/2019/10/llvm-clang-risc-v-now-supports-lto-michael-larabel-phoronix>>.
- [38] RISC-V Port Merged to Linux. Disponível em: <<https://groups.google.com/a/groups.riscv.org/g/sw-dev/c/2-u-c3kyZlc>>.
- [39] FEDORA - Architectures/RISC-V. Disponível em: <<https://fedoraproject.org/wiki/Architectures/RISC-V>>.
- [40] PORTING Alpine Linux to RISC-V. Disponível em: <<https://drewdevault.com/2018/12/20/Porting-Alpine-Linux-to-RISC-V.html>>.
- [41] ANDROID 10 ported to homegrown multi-core RISC-V system-on-chip by Alibaba biz, source code released. Disponível em: <[https://www.theregister.com/2021/01/21/android\\_riscv\\_port](https://www.theregister.com/2021/01/21/android_riscv_port)>.
- [42] MY Haiku RISC-V port progress. Disponível em: <<https://discuss.haiku-os.org/t/my-haiku-risc-v-port-progress/10663/85>>.
- [43] SEL4 is verified on RISC-V. Disponível em: <<https://microkerneldude.wordpress.com/2020/06/09/sel4-is-verified-on-risc-v>>.
- [44] TECHNOLOGIES, I. *Acquisition of MIPS Technologies completed.* Disponível em: <<https://web.archive.org/web/20131002214436/http://www.imgtec.com/news/Release/index.asp?NewsID=724>>.

- [45] BLOOMBERG. *Imagination Technologies Agrees to Takeover by Canyon Bridge*. Disponível em: <<https://www.bloomberg.com/news/articles/2017-09-22/imagination-technologies-agrees-to-takeover-by-canyon-bridge>>.
- [46] MIPS Acquired by AI Startup Wave Computing. Disponível em: <<https://www.top500.org/news/mips-acquired-by-ai-startup-wave-computing>>.
- [47] WAIT, What? MIPS Becomes RISC-V. Disponível em: <<https://www.eejournal.com/article/wait-what-mips-becomes-risc-v>>.
- [48] HARVARD Architecture. Disponível em: <<https://www.slideshare.net/CarmenUgay/harvard-architecture-12019907>>.
- [49] INTRODUCTION to "The First Draft Report on the EDVAC". Disponível em: <<https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf>>.
- [50] TERASIC. *DE1-SoC Board*. Disponível em: <[https://www.terasic.com.tw/attachment/archive/836/image/image\\_67\\_thumb.jpg](https://www.terasic.com.tw/attachment/archive/836/image/image_67_thumb.jpg)>.

# **ANEXOS**

# I. PROGRAMAS UTILIZADOS

- **Intel Quartus Prime Lite 18.1** - <https://fpgasoftware.intel.com/18.1/?edition=lite&platform=linux>
- **ModelSim Intel Edition** - [https://fpgasoftware.intel.com/?product=modelsim\\_ae#tabs-2](https://fpgasoftware.intel.com/?product=modelsim_ae#tabs-2)
- **RARS** - <https://github.com/TheThirdOne/rars/>
- **GTKWave** - <http://gtkwave.sourceforge.net/>
- **riscv-disassembler** - <https://github.com/michaeljclark/riscv-disassembler>
- **TeX Live** - <https://tug.org/texlive/>
- **Overleaf** - <https://www.overleaf.com/>
- **neovim** - <https://neovim.io/>
- **git** - <https://git-scm.com/>
- **bash** - <https://www.gnu.org/software/bash/>
- **sed** - <https://www.gnu.org/software/sed/>
- **Adobe Illustrator** - <https://www.adobe.com/products/illustrator/free-trial-download.html>
- **gimp** - <https://www.gimp.org/>
- **diagrams.net** - <https://app.diagrams.net/>

## II. DIAGRAMAS DAS MICROARQUITETURAS

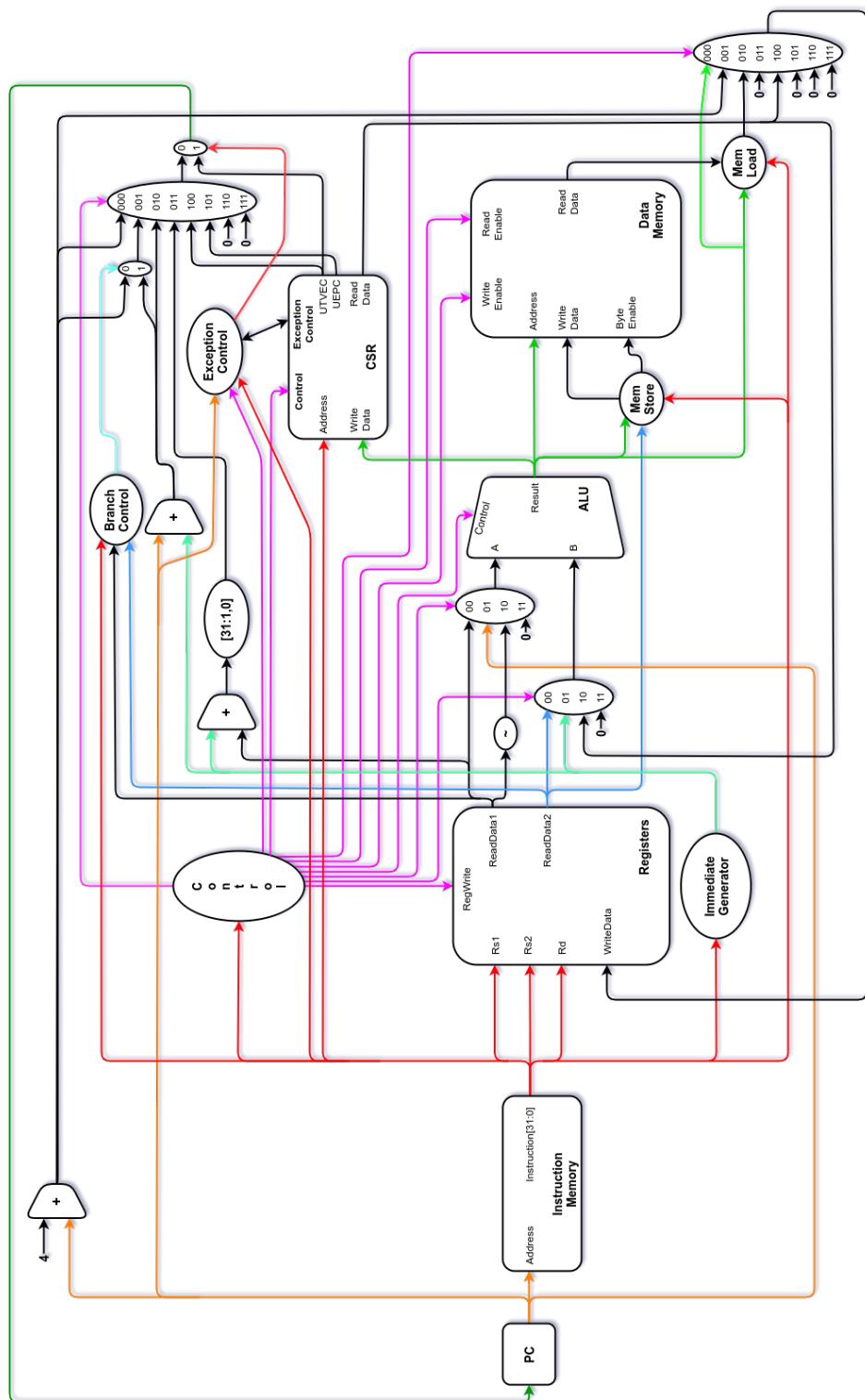


Figura II.1: Diagrama do uniciclo *RV32I e RV32IM*

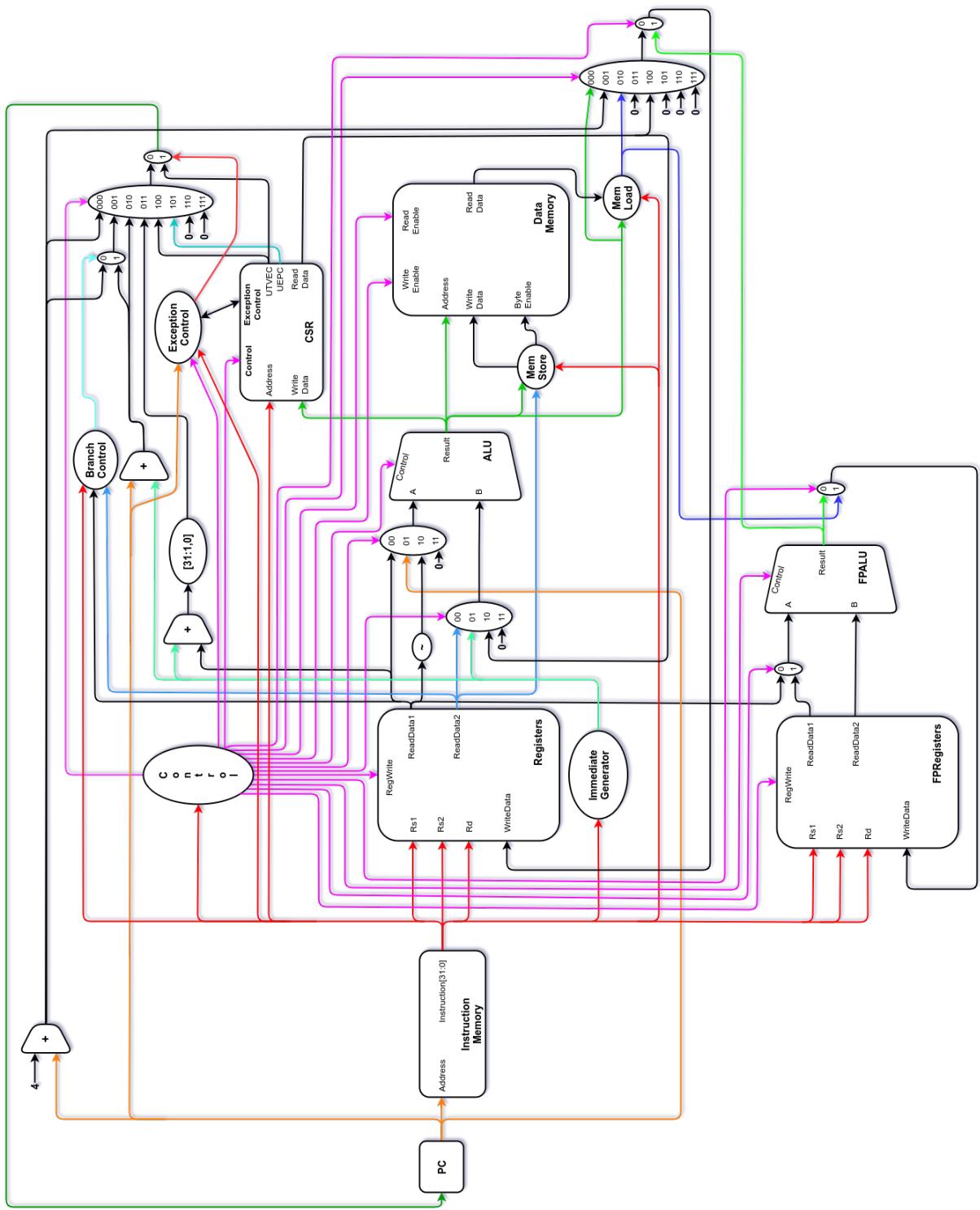


Figura II.2: Diagrama do uniciclo *RV32IMF*

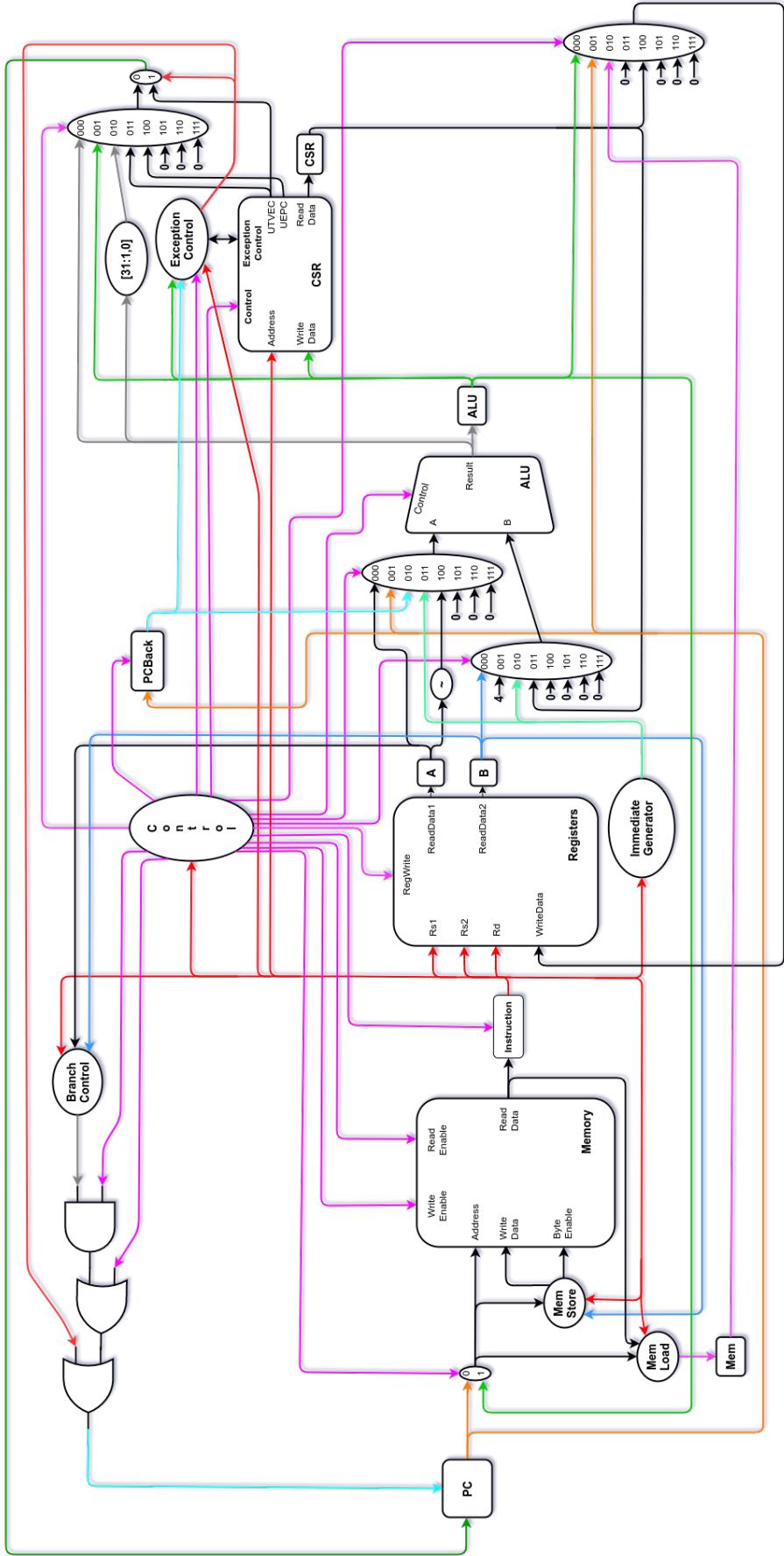


Figura II.3: Diagrama do multiciclo *RV32I e RV32IM*

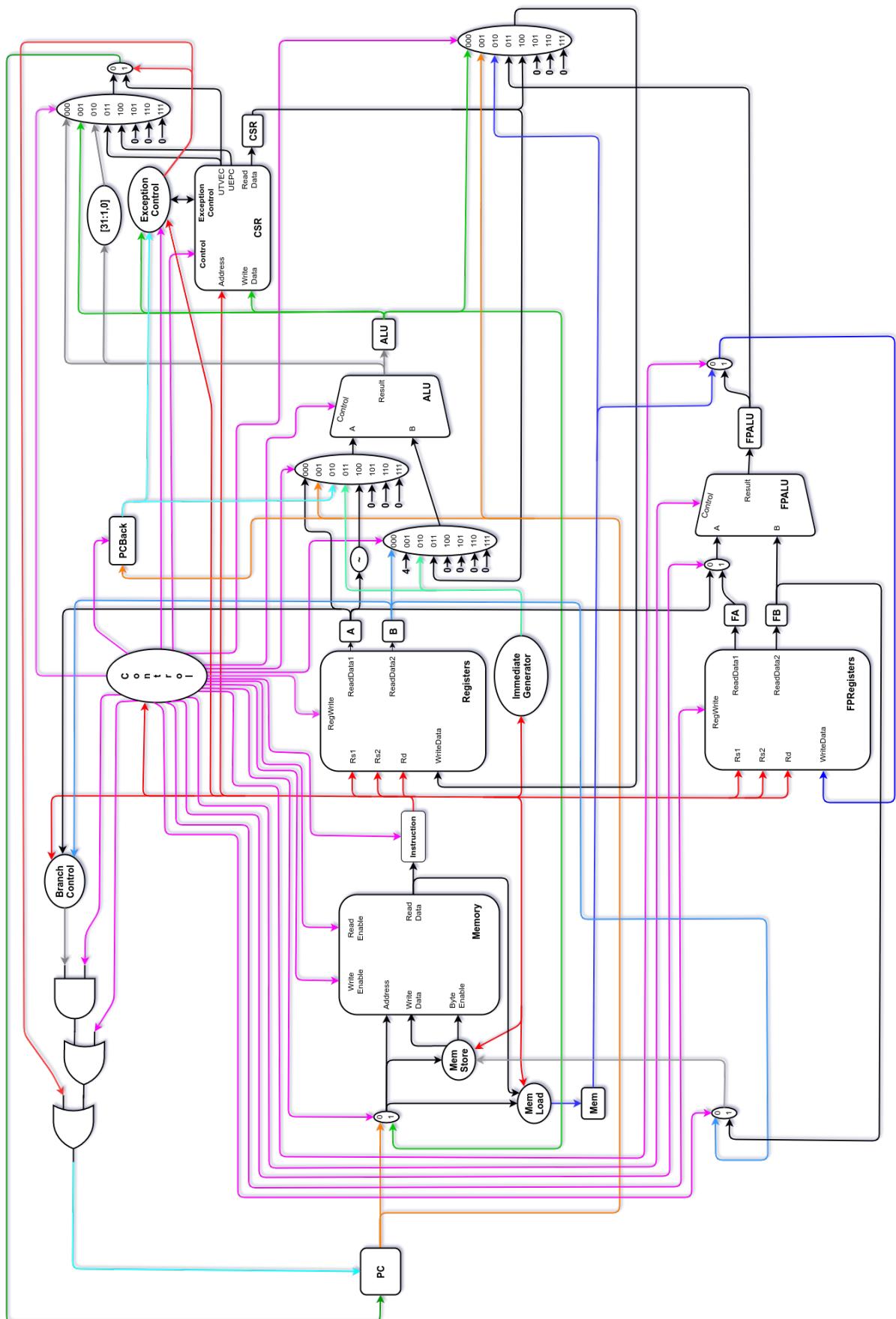


Figura II.4: Diagrama do multiciclo *RV32IMF*

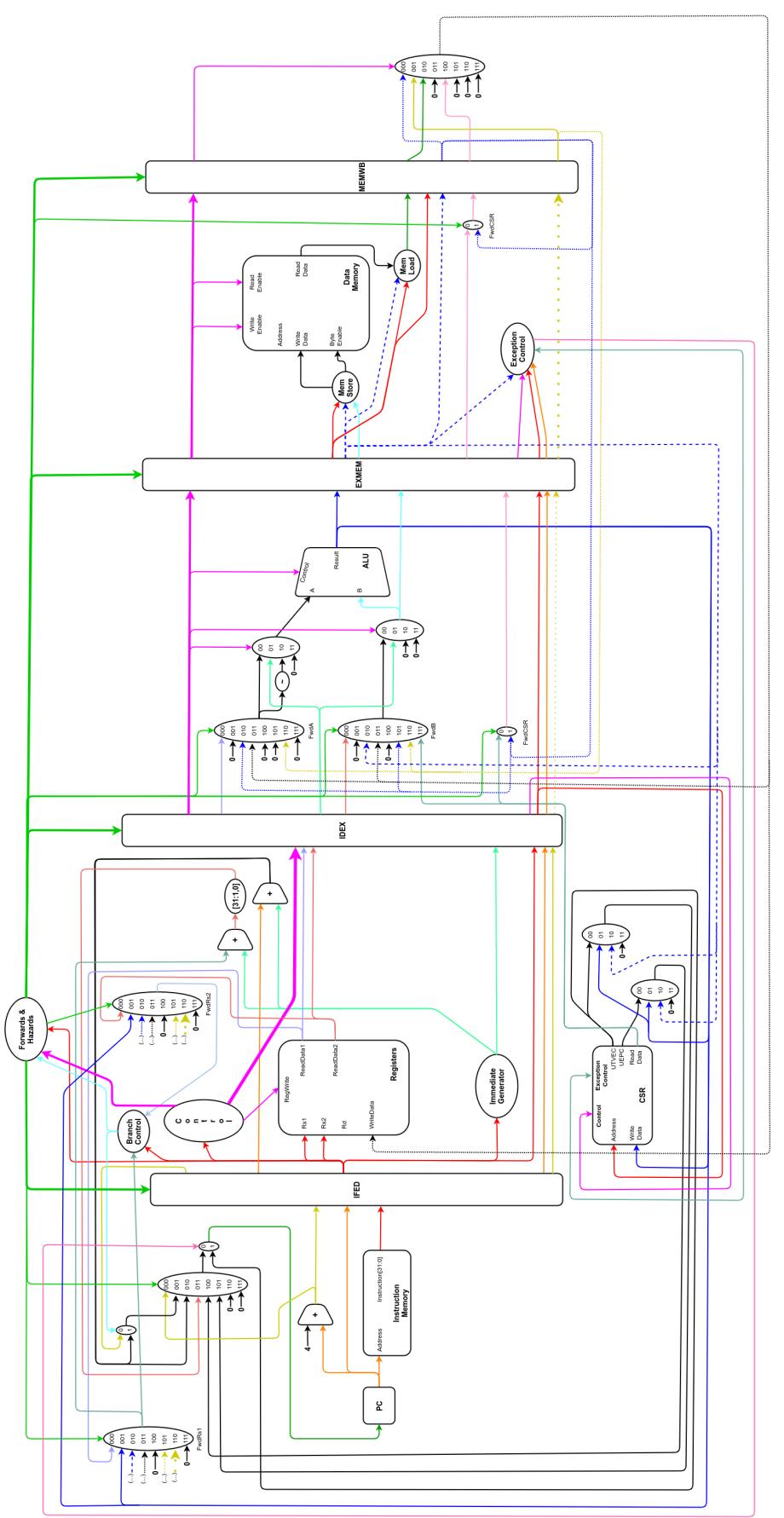


Figura II.5: Diagrama do pipeline *RV32I e RV32IM*

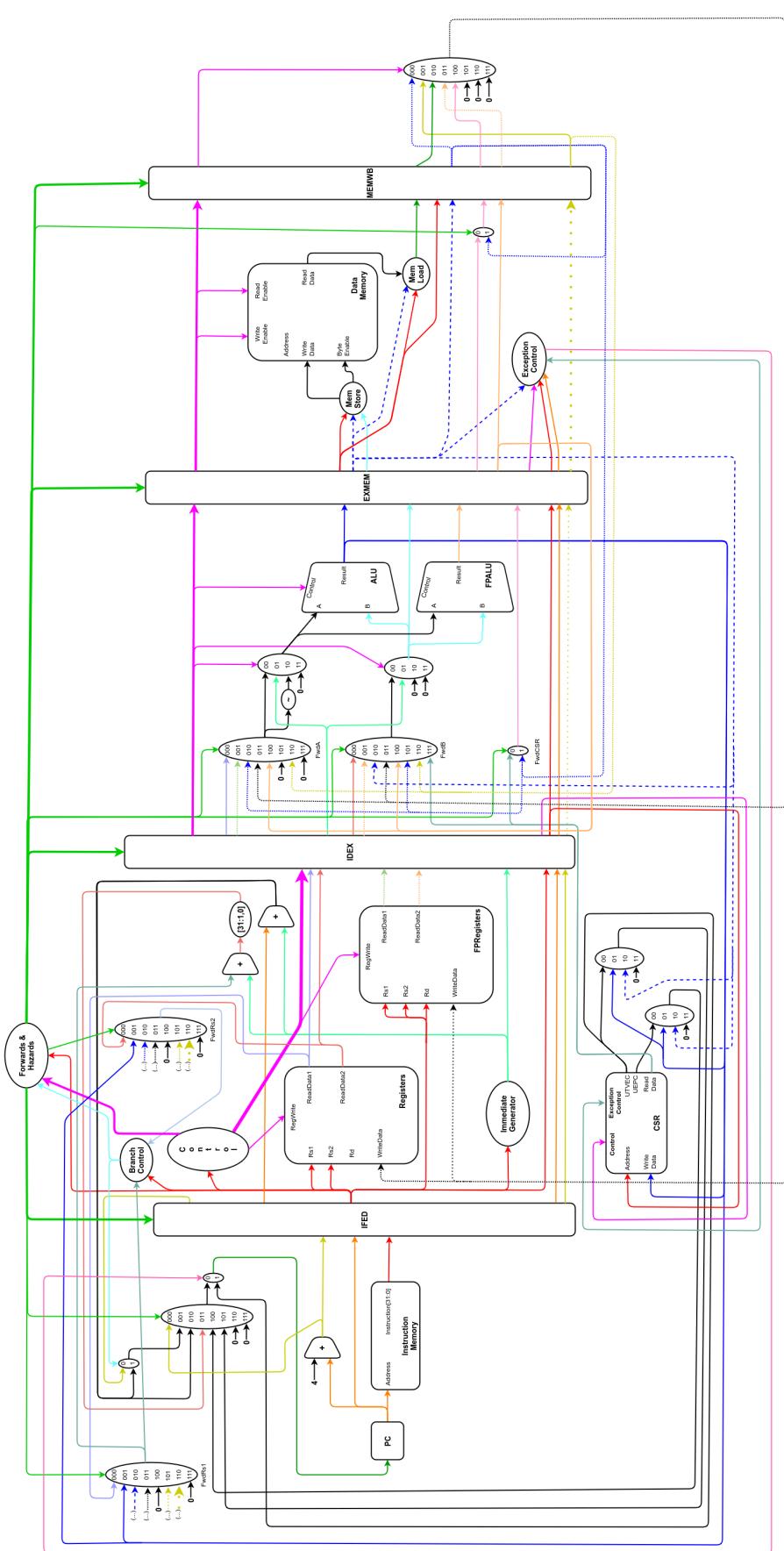


Figura II.6: Diagrama do pipeline *RV32IMF*