

Viviane Flor Park

169259

Projeto e Implementação de Processador ARM em Verilog e Sistema de Interface com FPGA

São José dos Campos

2025

Viviane Flor Park
169259

Projeto e Implementação de Processador ARM em Verilog e Sistema de Interface com FPGA

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Sérgio Ronaldo Barros dos Santos
Universidade Federal de São Paulo
Instituto de Ciência e Tecnologia

São José dos Campos
2025

Resumo

Este documento relata o projeto e implementação em Verilog de um processador ARM 32 bits e seu sistema de interface com FPGA. O projeto do processador descreve características gerais e detalha o conjunto de instruções adotado, como os diferentes tipos de instruções e respectivos caminhos de dados. É relatada também a implementação de todas as unidades funcionais do processador em Verilog, como *program counter*, memórias, banco de registradores, unidade lógica e aritmética, *current program status register*, unidade de controle etc, assim como a da integração por completo. Além disso, são descritos módulos de entrada e saída, os quais compõem um sistema de interface com FPGA. Para evidenciar o funcionamento desses elementos, são apresentadas simulações de formas de onda e de códigos teste.

Palavras-chaves: Processador ARM. Interface com FPGA. Conjunto de Instruções. Implementação em Verilog. Formas de onda. Códigos teste.

Listas de ilustrações

Figura 1 – Arquiteturas de von Neumann e Harvard. Fonte: (INSTITUTO FEDERAL DE SANTA CATARINA, 2020)	13
Figura 2 – Sufixos condicionais suportados pelo conjunto de instruções A32 das arquiteturas ARMv8/v9-A AArch32. Fonte: (ARM LIMITED, 2024) .	15
Figura 3 – Diferentes modos de endereçamento à memória. Fonte: (STALLINGS, 2017)	16
Figura 4 – <i>Field Programmable Gate Array</i> (FPGA). Fonte: A autora (2025).	17
Figura 5 – Visão geral da estrutura do processador. Fonte: A autora (2025).	19
Figura 6 – Caminho de dados das instruções do tipo D. Fonte: A autora (2025). .	21
Figura 7 – Caminho de dados das instruções do tipo B. Fonte: A autora (2025). .	22
Figura 8 – Caminho de dados das instruções do tipo M. Fonte: A autora (2025). .	23
Figura 9 – Caminho de dados das instruções do tipo A. Fonte: A autora (2025). .	24
Figura 10 – Código em Verilog da implementação do <i>Program Counter</i> (PC). Fonte: A autora (2025).	26
Figura 11 – Código em Verilog da implementação da memória de instruções. Fonte: A autora (2025).	27
Figura 12 – Código em Verilog da implementação da unidade de decodificação. Fonte: A autora (2025).	28
Figura 13 – Código em Verilog da implementação do banco de registradores. Fonte: A autora (2025).	29
Figura 14 – Código em Verilog da implementação da Unidade Lógica e Aritmética (ULA). Fonte: A autora (2025).	31
Figura 15 – Código em Verilog da implementação do <i>Current Program Status Register</i> (CPSR). Fonte: A autora (2025).	32
Figura 16 – Código em Verilog da implementação da memória de dados. Fonte: A autora (2025).	33
Figura 17 – Código em Verilog da implementação do multiplexador (MUX). Fonte: A autora (2025).	33
Figura 18 – Código em Verilog da implementação do somador ou <i>adder</i> . Fonte: A autora (2025).	34
Figura 19 – Código em Verilog da implementação da unidade de validação. Fonte: A autora (2025).	35
Figura 20 – Código em Verilog da implementação do módulo de integração entre memória de instruções e unidade de decodificação. Fonte: A autora (2025).	36

Figura 21 – Código em Verilog da implementação do módulo de integração entre banco de registradores e Unidade Lógica e Aritmética (ULA). Fonte: A autora (2025)	37
Figura 22 – Código em Verilog da implementação do módulo de entrada. Fonte: A autora (2025)	37
Figura 23 – Código em Verilog da implementação do módulo de saída. Fonte: A autora (2025)	38
Figura 24 – Códigos em Verilog das implementações dos módulos auxiliares à saída. Fonte: A autora (2025)	39
Figura 25 – Código em Verilog da implementação do seletor de <i>clock</i> . Fonte: A autora (2025)	40
Figura 26 – Código em Verilog da implementação da unidade de <i>debouncing</i> . Fonte: A autora (2025)	41
Figura 27 – Linhas 1 a 75 do código em Verilog da integração completa do processador. Fonte: A autora (2025)	42
Figura 28 – Linhas 76 a 152 do código em Verilog da integração completa do processador. Fonte: A autora (2025)	43
Figura 29 – Simulação de formas de onda do <i>Program Counter</i> (PC) gerada no <i>software Quartus Prime</i>	44
Figura 30 – Simulação de formas de onda da memória de instruções gerada no <i>software Quartus Prime</i>	45
Figura 31 – Simulação de formas de onda do módulo de integração da memória de instruções e unidade de decodificação gerada no <i>software Quartus Prime</i> .	46
Figura 32 – Simulação de formas de onda do banco de registradores gerada no <i>software Quartus Prime</i>	47
Figura 33 – Simulação de formas de onda da Unidade Lógica e Aritmética (ULA) gerada no <i>software Quartus Prime</i>	48
Figura 34 – Simulação de formas de onda do <i>Current Program Status Register</i> (CPSR) gerada no <i>software Quartus Prime</i>	49
Figura 35 – Simulação de formas de onda da memória de dados gerada no <i>software Quartus Prime</i>	49
Figura 36 – Simulação de formas de onda do multiplexador (MUX) gerada no <i>software Quartus Prime</i>	50
Figura 37 – Simulação de formas de onda do somador ou <i>adder</i> gerada no <i>software Quartus Prime</i>	50
Figura 38 – Simulação de formas de onda da unidade de validação gerada no <i>software Quartus Prime</i>	51

Figura 39 – Simulação de formas de onda do módulo de integração do banco de registradores, da Unidade Lógica e Aritmética (ULA) e do <i>Current Program Status Register</i> (CPSR) gerada no <i>software Quartus Prime</i>	52
Figura 40 – Simulação de formas de onda do módulo de entrada.	52
Figura 41 – Simulação de formas de onda do módulo de saída.	53
Figura 42 – Simulação de formas de onda do módulo de seleção de <i>clock</i>	53
Figura 43 – Códigos teste. Fonte: A autora (2025).	54
Figura 44 – Simulação de formas de onda do código teste de cálculo de média aritmética.	55
Figura 45 – Resultados do código de cálculo de média aritmética na placa FPGA. .	57
Figura 46 – Resultados do código de cálculo de área de triângulo na placa FPGA. .	58
Figura 47 – LED durante execução de instrução IN.	59

Lista de tabelas

Tabela 1 – Formato das instruções do tipo D. Fonte: A autora (2025).	20
Tabela 2 – Tipos, mnemônicos, op1 e descrições das instruções do tipo D. Fonte: A autora (2025).	20
Tabela 3 – Formato das instruções do tipo B. Fonte: A autora (2025).	21
Tabela 4 – Mnemônicos e descrições das instruções do tipo B. Fonte: A autora (2025).	21
Tabela 5 – Formato das instruções do tipo M. Fonte: A autora (2025).	22
Tabela 6 – Tipos, modos de endereçamento à memória, bits S, I e R e descrições das instruções do tipo M. Fonte: A autora (2025).	23
Tabela 7 – Formato das instruções do tipo A. Fonte: A autora (2025).	24
Tabela 8 – Mnemônicos, op1 e descrições das instruções do tipo A. Fonte: A autora (2025).	24
Tabela 9 – Descrições e pinagem dos pontos de comunicação do processador com o FPGA.	25
Tabela 10 – Operações codificadas pelo sinal de controle <i>CTRLOpULA</i> . Fonte: A autora (2025).	30
Tabela 11 – Instruções carregadas na memória de instruções, posições na memória e valores correspondentes em decimal. Fonte: A autora (2025).	45

Lista de abreviaturas e siglas

ALM	<i>Adaptable Logic Module</i>
BCD	<i>Binary-coded Decimal</i>
CISC	<i>Complex Instruction Set Computer</i>
CPSR	<i>Current Program Status Register</i>
CPU	<i>Central Processing Unit</i>
E/S	Entrada/Saída
FPGA	<i>Field Programmable Gate Array</i>
HDL	<i>Hardware Description Language</i>
IoT	<i>Internet of Things</i>
ISA	<i>Instruction Set Architecture</i>
LED	<i>Light Emitting Diode</i>
LR	<i>Link Register</i>
MSB	<i>Most Significant Bit</i>
MUX	Multiplexador
NZCV	<i>Negative, Zero, Carry and Overflow</i>
PC	<i>Program Counter</i>
RAM	<i>Random-access Memory</i>
RISC	<i>Reduced Instruction Set Computer</i>
ROM	<i>Read-only Memory</i>
SP	<i>Stack Pointer</i>
ULA	Unidade Lógica e Aritmética

Sumário

1	INTRODUÇÃO	10
2	OBJETIVOS	11
2.1	Geral	11
2.2	Específicos	11
3	FUNDAMENTAÇÃO TEÓRICA	12
3.1	Computadores, Arquitetura e Organização	12
3.2	Arquitetura de Von Neumann vs. Harvard	12
3.3	Arquitetura do Conjunto de Instruções	13
3.3.1	RISC vs. CISC	13
3.4	Arquitetura ARM	14
3.4.1	Execução Condicional de Instruções	14
3.5	Modos de Endereçamento	15
3.5.1	Imediato	15
3.5.2	Direto	16
3.5.3	Indireto por Registrador	16
3.5.4	Deslocamento	16
3.6	Linguagem de Descrição de Hardware	17
3.7	Field Programmable Gate Array (FPGA)	17
4	DESENVOLVIMENTO	18
4.1	Características Gerais do Processador	18
4.2	Execução Condicional de Instruções	19
4.3	Conjunto de Instruções e Caminhos de Dados	19
4.3.1	Instruções de Processamento de Dados: Tipo D	19
4.3.2	Instruções de Desvio: Tipo B	21
4.3.3	Instruções de Acesso à Memória: Tipo M	22
4.3.4	Instruções Alternativas: Tipo A	23
4.4	Interface com FPGA	25
4.5	Implementação em Verilog	25
4.5.1	Program Counter (PC)	25
4.5.2	Memória de Instruções	26
4.5.3	Unidade de Decodificação	27
4.5.4	Banco de Registradores	28
4.5.5	Unidade Lógica e Aritmética (ULA)	29

4.5.6	<i>Current Program Status Register (CPSR)</i>	31
4.5.7	Memória de Dados	32
4.5.8	Multiplexador (MUX)	33
4.5.9	<i>Adder</i>	34
4.5.10	Unidade de Validação	34
4.5.11	Integração Memória de Instruções e Unidade de Decodificação	36
4.5.12	Integração Banco de Registradores e ULA	36
4.5.13	Entrada e Saída (E/S)	37
4.5.14	Unidade de <i>Debouncing</i> e Seletor de <i>Clock</i>	39
4.5.15	Integração Completa	41
5	RESULTADOS E DISCUSSÃO	44
5.1	<i>Program Counter (PC)</i>	44
5.2	Memória de Instruções	44
5.3	Unidade de Decodificação	45
5.4	Banco de Registradores	46
5.5	Unidade Lógica e Aritmética (ULA)	48
5.6	<i>Current Program Status Register (CPSR)</i>	48
5.7	Memória de Dados	49
5.8	Multiplexador (MUX)	49
5.9	<i>Adder</i>	50
5.10	Unidade de Validação	50
5.11	Integração Banco de Registradores e ULA	51
5.12	Entrada e Saída (E/S)	52
5.13	Seletor de <i>Clock</i>	53
5.14	Integração Completa	53
5.14.1	Sem Interface com FPGA	54
5.14.2	Com Interface com FPGA	56
6	CONSIDERAÇÕES FINAIS	60
	REFERÊNCIAS	61

1 Introdução

Um sistema computacional é um conjunto integrado de componentes eletrônicos e lógicos que trabalham em conjunto para processar, armazenar e transmitir informações. Ele é composto principalmente pelo *hardware*, como o processador, a memória e os dispositivos de entrada e saída, e pelo *software*, que inclui os sistemas operacionais e os aplicativos que gerenciam e executam as tarefas desejadas. Eles estão presentes em praticamente todos os aspectos da vida moderna, desde computadores pessoais e *smartphones* até sistemas embarcados em eletrodomésticos, veículos e equipamentos industriais, o que sempre reforça a necessidade de sistemas compactos, rápidos e energeticamente eficientes.

Nesse contexto, o estudo da arquitetura e organização de computadores torna-se fundamental, pois fornece o conhecimento necessário para entender como um processador interpreta e executa instruções, como a memória é gerenciada e como os dados circulam pelo sistema. Uma das arquiteturas mais relevantes atualmente é a ARM, que se destaca por seu *design* eficiente e baixo consumo de energia. Muito utilizada em dispositivos móveis e embarcados, a arquitetura ARM é um exemplo de como decisões de projeto arquitetônico podem impactar diretamente o desempenho, a autonomia e o custo de um sistema computacional.

Este projeto se baseia na teoria da arquitetura e organização de computadores para o desenvolvimento de um processador ARM adaptado. Primeiramente, são definidas características relevantes do processador, como número de registradores, estrutura, conjunto de instruções etc. Em seguida, a implementação do processador requer a utilização de linguagem de descrição de *hardware*, em específico o Verilog, para o projeto das unidades funcionais e do *software* Quartus Prime para a simulação de formas de onda para a verificação de funcionamento. Por último, é descrita a implementação de um sistema de interface com uma placa FPGA.

2 Objetivos

2.1 Geral

O projeto visa trabalhar conceitos teóricos de arquitetura e organização de computadores por meio do projeto e implementação de um processador em linguagem de descrição de *hardware*, além da construção de um sistema de interface com FPGA.

2.2 Específicos

São propostos também objetivos específicos a se cumprir durante o desenvolvimento deste projeto, conforme listados a seguir.

- Definição de um conjunto de instruções abrangente e flexível;
- Definição de formatos de instruções e seus modos de endereçamento à memória;
- Definição da estrutura do processador e caminhos de dados das instruções;
- Adaptação da arquitetura ARM conforme necessário;
- Implementação das unidades funcionais do processador em Verilog;
- Validação do funcionamento das unidades funcionais implementadas por meio da simulação de formas de onda no *software* Quartus Prime;
- Instanciação e integração das unidades funcionais implementadas;
- Validação do funcionamento da unidade integrada do processador por meio da simulação de formas de onda no *software* Quartus Prime;
- Desenvolvimento de um sistema de interface do processador com FPGA;
- Validação do sistema de interface desenvolvido por meio de programas teste.

3 Fundamentação Teórica

Este capítulo cobre toda a teoria necessária para a descrição do projeto do processador. As seções 3.1, 3.2 e 3.3 discutem conceitos iniciais de computadores, classificações de arquiteturas e organização. Já na seção 3.4 a arquitetura ARM é descrita com mais detalhes, e na seção 3.5 são discutidos diferentes modos de endereçamento à memória. As seções 3.6 e 3.7 descrevem as ferramentas utilizadas para o desenvolvimento prático do projeto.

3.1 Computadores, Arquitetura e Organização

Um computador é qualquer dispositivo que contenha um processador (ou CPU, *Central Processing Unit*, do inglês), memória principal, dispositivos de entrada e saída e sistemas de interconexão de todos esses componentes. Normalmente são atribuídas a ele quatro funções gerais: processamento, armazenamento e movimentação de dados, além do controle e gerenciamento de todos os seus recursos (STALLINGS, 2017).

No projeto de um computador, todos os atributos que impactam diretamente na lógica de um programa, ou seja, tudo que é de alguma forma relevante a um programador é designado arquitetura. São considerados elementos da arquitetura, por exemplo, as instruções que um computador pode executar, a quantidade de registradores de propósito geral no processador e a quantidade de bits designada a determinados dados. Por outro lado, existem atributos do sistema que dizem respeito às interconexões de unidades funcionais, o que não impacta diretamente no jeito que se programa. Estes elementos fazem parte da organização de um computador.

3.2 Arquitetura de Von Neumann vs. Harvard

Duas arquiteturas famosas, principalmente no que diz respeito aos sistemas de memória adotados, são a de von Neumann e a de Harvard. A primeira foi proposta pelo húngaro John von Neumann em 1945 e é amplamente utilizada em computadores pessoais e servidores. Já a segunda surgiu a partir de projetos de computadores para fins específicos, como os primeiros sistemas de controle militar, e hoje é comum em microcontroladores e sistemas embarcados.

A principal diferença entre essas duas arquiteturas está na forma como instruções e dados são armazenados e acessados. Na arquitetura de von Neumann, ambos compartilham a mesma memória e se utilizam de um barramento único. Isso, apesar de simplificar o projeto

do *hardware*, também pode causar gargalos por causa da disputa pelo barramento. Já na arquitetura de Harvard, instruções e dados ficam em memórias separadas, com barramentos distintos, permitindo acessos simultâneos e maior eficiência em certas aplicações. Essa diferença torna a arquitetura de Harvard especialmente vantajosa em sistemas que exigem alto desempenho com recursos limitados, como dispositivos embarcados. A Figura 1 mostra um esquemático de como as duas arquiteturas funcionam.

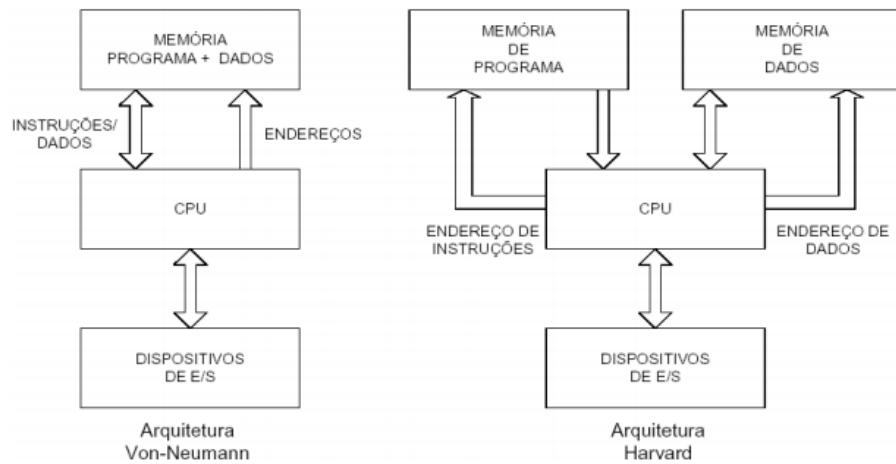


Figura 1 – Arquiteturas de von Neumann e Harvard. Fonte: ([INSTITUTO FEDERAL DE SANTA CATARINA, 2020](#))

3.3 Arquitetura do Conjunto de Instruções

Uma arquitetura do conjunto de instruções (ou ISA, *Instruction Set Architecture*, do inglês) é uma parte crucial da arquitetura de um computador, e atua como interface entre o *hardware* e o *software* (STALLINGS, 2017). Ela define tudo a respeito do conjunto de instruções ao qual um processador tem suporte e, consequentemente, tem impacto em todo o seu âmbito programável. Alguns exemplos de ISAs famosas são MIPS, x86 e ARM.

3.3.1 RISC vs. CISC

ISAs podem ser classificadas em *Reduced Instruction Set Computer* (RISC) ou *Complex Instruction Set Computer* (CISC). A primeira descreve arquiteturas de instruções simples e de formato fixo, primordialmente com referências à memória do tipo *load-store*, ou seja, que utilizam registradores como intermediários. Por esse motivo, geralmente contam com um grande banco de registradores e são estruturadas de acordo com a arquitetura de Harvard. Já a segunda abrange arquiteturas de instruções mais complexas e de formato variável, nas quais qualquer instrução pode fazer referência à memória. Estas seguem, normalmente, os padrões da arquitetura de von Neumann (TOSTA, 2024).

3.4 Arquitetura ARM

A família das arquiteturas do conjunto de instruções ARM foi desenvolvida inicialmente em meados dos anos 80 pela empresa britânica ARM Holdings. Se trata de uma família de arquiteturas RISC muito utilizada atualmente, principalmente em *smartphones*, sistemas embarcados e dispositivos de IoT (internet das coisas ou, do inglês, *Internet of Things*), pela sua grande eficiência energética ([ARM LIMITED, 2025](#)).

Esta ISA tem suporte para instruções e registradores de 32 bits e, em versões mais recentes, também para 64 bits. Em específico para a arquitetura de 32 bits, existem mecanismos de suporte para conjuntos de instruções de 16 e 32 bits. Exemplos disso são os conjuntos Thumb e ARM padrão na versão ARMv5 ([ARM LIMITED, 2000](#)) e os conjuntos T32 e A32 nas versões ARMv8/v9-A ([ARM LIMITED, 2024](#)).

Ainda para a arquitetura de 32 bits, existem 15 registradores de propósito geral. Entre estes, R14 e R13 são respectivamente usados como *Stack Pointer* (SP), para o manuseio de pilha, e *Link Register* (LR), para a chamada de funções. Existem também dois registradores especiais de uso dedicado: o *Program Counter* (PC) e o *Current Program Status Register* (CPSR). O primeiro guarda o endereço da instrução a ser executada, enquanto o segundo guarda informações a respeito do programa sendo executado pelo processador. O CPSR, além de manter controle sobre todos os 9 modos de processador (para as arquiteturas ARMv8/v9-A), também é essencial para a execução condicional de instruções, que será aprofundada na seção [3.4.1](#).

3.4.1 Execução Condisional de Instruções

A arquitetura ARM se destaca pela sua lógica de executar condicionalmente instruções. Por meio dela, qualquer instrução pode estar sujeita a uma condição predefinida que, se validada pelo processador, permite e, caso contrário, inibe sua execução.

As operações realizadas pela Unidade Lógica e Aritmética (ULA) podem produzir resultados negativos, nulos, que resultam em *carry* ou em *overflow*. Essas informações são então guardadas por 4 bits especiais do CPSR, conhecidos como *flags* NZCV (do inglês, *negative, zero, carry and overflow*), que servem como parâmetros para a validação de instruções.

A imposição de uma condição é feita por meio da adição de um sufixo condicional mnemônico a uma instrução, o qual pressupõe um estado específico das *flags*. Esse sufixo é então convertido a um campo específico da instrução chamado de campo *cond*, o qual é comparado com as *flags* do CPSR. Essa comparação gera, então, um sinal que decide pela execução ou inibição da instrução.

Por exemplo, a Figura [2](#) mostra os sufixos condicionais mnemônicos adotados pelo

conjunto de instruções A32 das arquiteturas ARMv8/v9-A, suas respectivas descrições, campos *cond* e estados das *flags* NZCV a fim de serem validadas pelo processador.

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point)^a	Condition flags
0000	EQ	Equal	Equal	$Z == 1$
0001	NE	Not equal	Not equal, or unordered	$Z == 0$
0010	CS ^b	Carry set	Greater than, equal, or unordered	$C == 1$
0011	CC ^c	Carry clear	Less than	$C == 0$
0100	MI	Minus, negative	Less than	$N == 1$
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	$N == 0$
0110	VS	Overflow	Unordered	$V == 1$
0111	VC	No overflow	Not unordered	$V == 0$
1000	HI	Unsigned higher	Greater than, or unordered	$C == 1$ and $Z == 0$
1001	LS	Unsigned lower or same	Less than or equal	$C == 0$ or $Z == 1$
1010	GE	Signed greater than or equal	Greater than or equal	$N == V$
1011	LT	Signed less than	Less than, or unordered	$N != V$
1100	GT	Signed greater than	Greater than	$Z == 0$ and $N == V$
1101	LE	Signed less than or equal	Less than, equal, or unordered	$Z == 1$ or $N != V$
1110	None (AL) ^d	Always (unconditional)	Always (unconditional)	Any

Figura 2 – Sufixos condicionais suportados pelo conjunto de instruções A32 das arquiteturas ARMv8/v9-A AArch32. Fonte: ([ARM LIMITED, 2024](#))

3.5 Modos de Endereçamento

Ao se tratar de um conjunto de instruções, surge a questão de como codificar seus operandos e operações. Em especial, instruções de acesso à memória, que operam com endereços de memória a fim de guardar ou buscar operandos, empregam algumas técnicas diferentes para tal, habitualmente chamadas de modos de endereçamento à memória. Essas técnicas podem ser utilizadas para aumentar a flexibilidade de referência à memória ou para a diminuição da complexidade de cálculo de endereços ([STALLINGS, 2017](#)). Algumas destas são detalhadas nas seções 3.5.1 a 3.5.4.

3.5.1 Imediato

O modo mais simples de endereçamento é o imediato, no qual, conforme visto na Figura 3 (a), o operando em questão está descrito na própria instrução, no campo destinado a um endereço de memória. A vantagem dessa técnica é que a memória não é acessada, o que pode tornar sua execução mais rápida. Por outro lado, o campo onde está o operando é limitado, o que também limita sua magnitude.

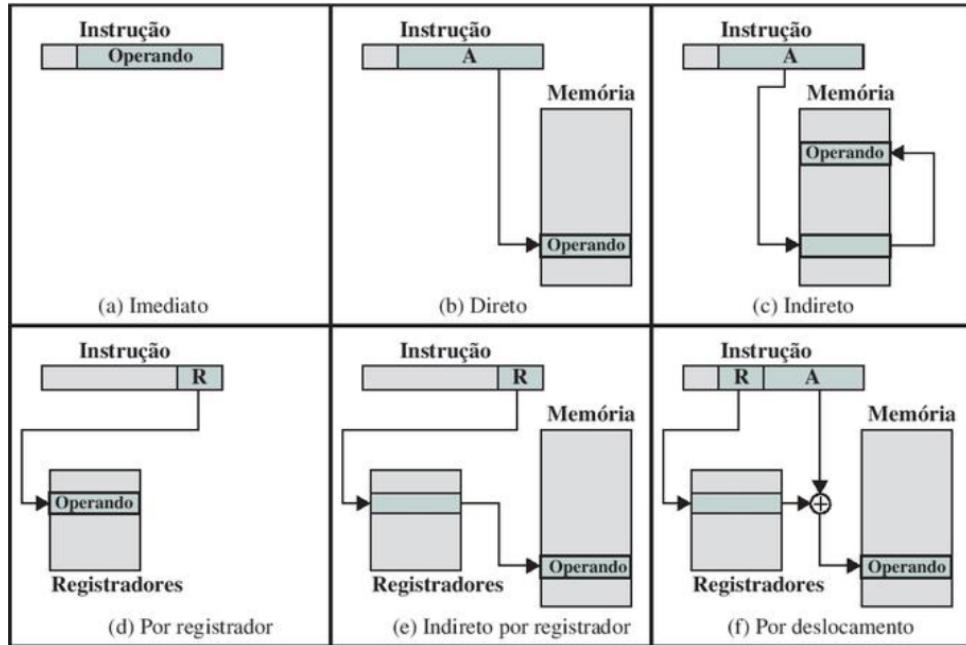


Figura 3 – Diferentes modos de endereçamento à memória. Fonte: ([STALLINGS, 2017](#))

3.5.2 Direto

No modo de endereçamento direto, um campo da instrução contém um endereço da própria memória principal (A) onde está ou para onde vai o operando. A vantagem dessa técnica é que não há nenhuma complexidade no cálculo do endereço, visto que ele é fornecido diretamente. Por outro lado, o campo da instrução onde está o endereço pode limitar o intervalo da memória principal que pode ser acessada.

3.5.3 Indireto por Registrador

A técnica do endereçamento indireto por registrador guarda o endereço de destino ou de origem do operando em um registrador, o qual é referenciado pela instrução (R). Esta é uma mistura dos modos de endereçamento indireto e por registrador, conforme observado na Figura 3. A utilização de um registrador como mediador amplia as possibilidades de endereços e evita o acesso duplo à memória, comparado ao endereçamento indireto.

3.5.4 Deslocamento

O endereçamento por deslocamento soma um valor imediato a um endereço contido em um registrador, criando, assim, um novo endereço efetivo. Ambos o imediato (A) e o registrador (R) são explicitados na instrução. Uma vantagem desse método é o acesso a endereços subsequentes a um endereço base.

3.6 Linguagem de Descrição de Hardware

Uma linguagem de descrição de *hardware* (HDL, do inglês *hardware description language*), como o Verilog ou o VHDL, é uma forma de descrever circuitos digitais por meio de código textual, permitindo modelar o comportamento e a estrutura de sistemas eletrônicos. Diferente das linguagens de programação tradicionais, as HDLs são usadas para projetar *hardware* real, como processadores, controladores e sistemas embarcados. Com elas, é possível simular, testar e, posteriormente, sintetizar os circuitos para implementação física.

3.7 Field Programmable Gate Array (FPGA)

Um *Field Programmable Gate Array*, ou FPGA, conforme visto na Figura 4, é um circuito integrado multifacetado projetado para ser programável após sua fabricação (IBM, 2024). Esse dispositivo é construído de maneira que módulos de lógica adaptável (ou ALM, *adaptable logic module*), compostos por portas lógicas e *flip-flops*, possam ser configurados e reconfigurados da maneira desejada com outros blocos especializados, como *random-access memory* (RAM), por meio de sinais elétricos (INTEL, 2025).

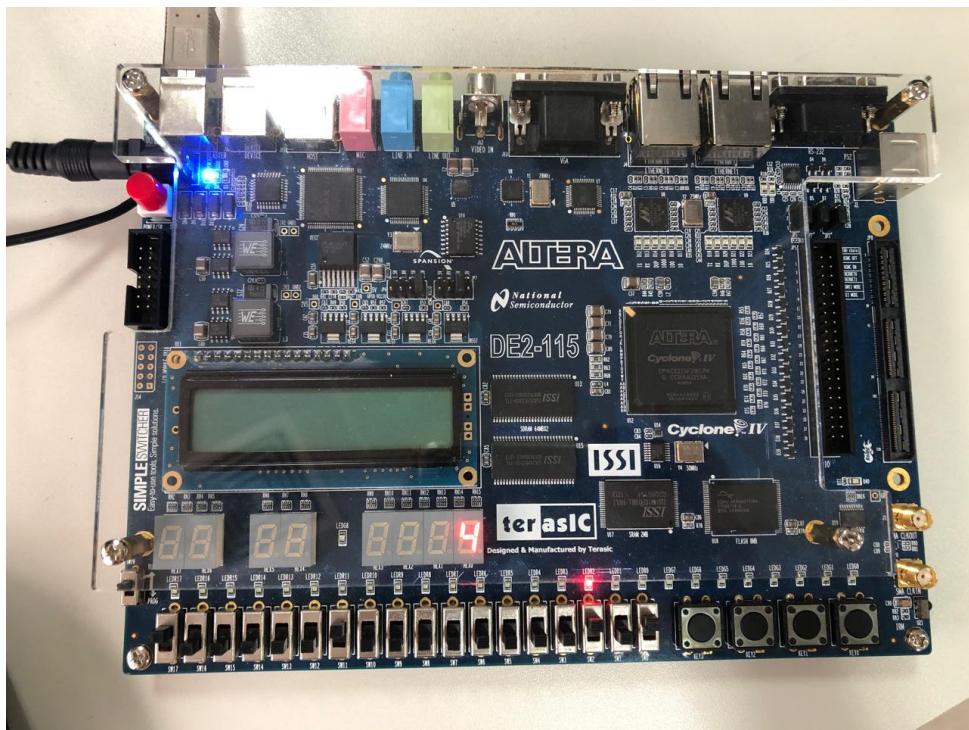


Figura 4 – *Field Programmable Gate Array* (FPGA).

Fonte: A autora (2025).

4 Desenvolvimento

Neste capítulo são descritos os detalhes do projeto do processador. As seções [4.1](#) e [4.2](#) descrevem as características gerais do processador e detalhes a respeito da execução condicional de instruções, respectivamente, enquanto a seção [4.3](#) se aprofunda nas características do conjunto de instruções adotado e nos possíveis caminhos de dados. A seção [4.5](#) descreve a implementação em Verilog de cada unidade funcional do processador.

4.1 Características Gerais do Processador

O processador foi projetado de acordo com a arquitetura ARM, salvo algumas mudanças necessárias para sua implementação. Assim, como se trata de uma arquitetura RISC, foi definido um tamanho fixo para as instruções de 32 bits.

Este projeto descreve um processador monociclo que segue os padrões da arquitetura de Harvard, ou seja, memória de programa e de dados separadas. Para fins desta implementação, a memória de programa guarda uma palavra de 32 bits por linha e o banco de registradores conta com 32 registradores de propósito geral, todos de 32 bits. Conforme visto na seção [3.4](#), isso diverge da maneira que a arquitetura é geralmente implementada. Entre os registradores de propósito geral, o registrador R31 foi reservado para uso como LR. Também foram projetados dois registradores de uso dedicado: o PC, de 32 bits, e o CPSR, de 4 bits.

Conforme mostra a estrutura geral do processador na Figura [5](#), ele conta com algumas unidades funcionais diferentes, como PC, memória de instruções, unidade de decodificação, banco de registradores, ULA, CPSR, memória de dados, unidade de validação, unidade de controle, multiplexadores e um *adder*. Em específico para o desenvolvimento do sistema de interface com FPGA, foram criados módulos adicionais como entrada, saída, um seletor de *clock* e uma unidade de *debouncing*. Detalhes a respeito dos propósitos, implementação, funcionamento e interligação de cada uma dessas unidades podem ser obtidos na seção [4.5](#).

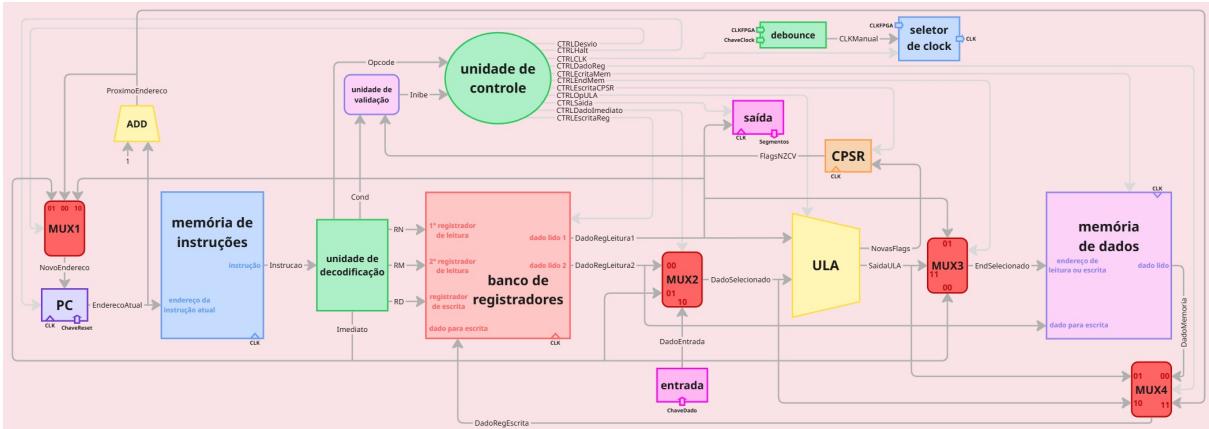


Figura 5 – Visão geral da estrutura do processador. Fonte: A autora (2025).

4.2 Execução Condicional de Instruções

O processador foi projetado para implementar a lógica da execução condicional de todas as instruções, de acordo com o que foi visto na seção 3.4.1. Os sufixos condicionais utilizados são os mesmos da Figura 2 e, portanto, o campo *cond* ocupa 4 bits.

O CPSR guarda somente as *flags* NZCV, dispensando a necessidade de bits para controle de modos de processador. Ainda para a implementação dessa lógica, é preciso fazer também o cálculo e a validação dessas *flags*. O primeiro é feito dentro da própria ULA e é discutido com mais profundidade na seção 4.5.5, enquanto a segunda é feita pela unidade de validação, conforme visto na Figura 5. Esse módulo compara o campo *cond* da instrução com as *flags* do CPSR e gera um sinal que é alimentado à unidade de controle. Assim, ela consegue permitir ou inibir a execução dessa instrução.

4.3 Conjunto de Instruções e Caminhos de Dados

O conjunto de instruções adotado foi adaptado dos conjuntos ARM padrão (ARM LIMITED, 2000) e A32 (ARM LIMITED, 2024) para a arquitetura de 32 bits. Nesta adaptação, as instruções foram subdivididas em instruções de processamento de dados, de desvio, de acesso à memória e alternativas, descritas nas seções 4.3.1 a 4.3.4.

4.3.1 Instruções de Processamento de Dados: Tipo D

As instruções de processamento de dados realizam operações aritméticas, lógicas, de movimentação ou de comparação entre valores. Estas serão referenciadas como instruções de tipo D, e são codificadas de acordo com o formato da Tabela 1.

Tabela 1 – Formato das instruções do tipo D. Fonte: A autora (2025).

Formato	<i>cond</i>	op0 (00)	I	op1	RN	RD	RM/Immediato
Bits	31:28	27:26	25	24:20	19:15	14:10	9:0

O formato de instruções do tipo D pode ser resumido em três partes principais: o campo *cond* (31:28), o *opcode* (27:20) e os bits de dados (19:0). O *opcode* desse tipo de instrução conta com 8 bits, e pode ser subdividido em 3 partes: o op0, o bit I e o op1. O campo op0 indica o tipo de instrução, e assume valor 00 para instruções do tipo D. O bit I é resetado se os bits 9 a 0 contêm o endereço de um registrador (9:5), e setado caso contenham um valor imediato (9:0). A individualização das instruções é realizada pelo campo op1, que destaca a operação a ser realizada. A Tabela 2 lista todas as instruções de tipo D, seus mnemônicos, descrições e respectivos campos op1.

Tabela 2 – Tipos, mnemônicos, op1 e descrições das instruções do tipo D.
Fonte: A autora (2025).

Tipo	Mnemônico	op1	Descrição
Aritméticas	ADD	00000	$RD \leq RN + RM/Im$
	ADDS	00001	$RD \leq RN + RM/Im$ Altera <i>flags</i>
	SUB	00010	$RD \leq RN - RM/Im$
	SUBS	00011	$RD \leq RN - RM/Im$ Altera <i>flags</i>
	RSB	00100	$RD \leq RM/Im - RN$ Altera <i>flags</i>
	MUL	00101	$RD \leq RN \cdot RM/Im$
Lógicas	UDIV	00110	$RD \leq RN \div RM/Im$
	NOT	00111	$RD \leq RM/Im$
	AND	01000	$RD \leq RN \text{ and } RM/Im$
	ORR	01001	$RD \leq RN \text{ or } RM/Im$
Movimentação	EOR	01010	$RD \leq RN \text{ xor } RM/Im$
	MOV	01011	$RD \leq RN/Im$
Comparação	CMP	01100	$RD \leq RN - RM/Im$ Altera <i>flags</i>

Vale ressaltar que a implementação do bit I dispensa a necessidade de instruções específicas para tratamento de imediatos, como ADDI ou SUBI.

Instruções do tipo D seguem os possíveis caminhos de dados ilustrados na Figura 6.

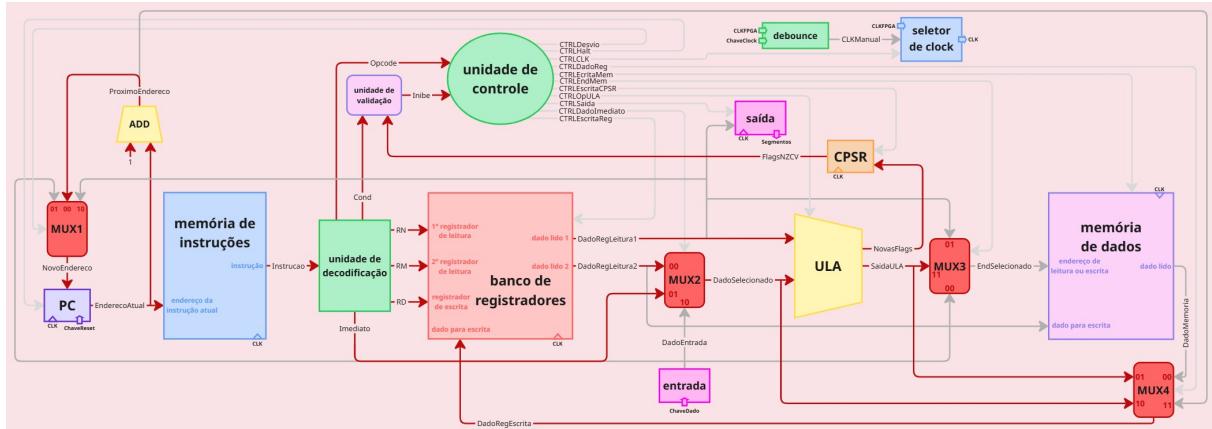


Figura 6 – Caminho de dados das instruções do tipo D. Fonte: A autora (2025).

4.3.2 Instruções de Desvio: Tipo B

As instruções de desvio ou *branch*, do inglês, realizam saltos. A lógica da execução condicional de instruções também se aplica a instruções de desvio, apesar de, a princípio, operarem de forma incondicional. Instruções desse tipo serão chamadas também de instruções de tipo B, e estão codificadas de acordo com a Tabela 3.

Tabela 3 – Formato das instruções do tipo B. Fonte: A autora (2025).

Formato	<i>cond</i>	<i>op0 (01)</i>	<i>op1</i>	RN	Imediato
Bits	31:28	27:26	25:24	23:19	18:0

Estas instruções possuem campo *cond* (31:28) e *opcode* de 4 bits (27:24). O op0 assume valor 01 e indica, desta vez, uma instrução do tipo B. Os bits restantes do *opcode* compõe o op1, campo que distingue cada instrução e sua respectiva funcionalidade. A Tabela 4 lista todas as instruções de desvio, seus mnemônicos, op1 e descrições.

Tabela 4 – Mnemônicos e descrições das instruções do tipo B. Fonte: A autora (2025).

Mnemônico	op1	Descrição
B	00	PC \leq Imediato Pula para <i>label</i>
BX	01	PC \leq RN
BL	10	RD (LR) \leq PC + 1 PC \leq Imediato Pula para <i>label</i>

Os bits 23 a 19 de uma instrução de tipo B codificam o registrador RN que contém o endereço de desvio, no caso da instrução BX. Para as instruções B e BL, está codificado um imediato nos bits 18 a 0, que representa o endereço absoluto de um *label* na memória de

programa, que é o endereço de desvio. Instruções do tipo B seguem os possíveis caminhos de dados ilustrados da Figura 7.

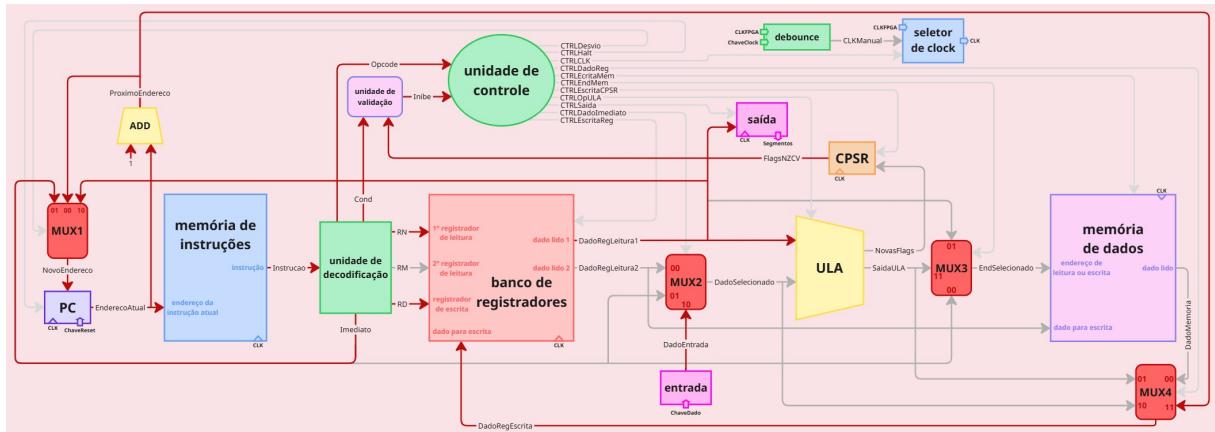


Figura 7 – Caminho de dados das instruções do tipo B. Fonte: A autora (2025).

4.3.3 Instruções de Acesso à Memória: Tipo M

Instruções de acesso à memória carregam dados da memória principal em registradores e vice-versa. Estas serão referenciadas como instruções do tipo M, e estão codificadas de acordo com a Tabela 5.

Tabela 5 – Formato das instruções do tipo M. Fonte: A autora (2025).

Formato	<i>cond</i>	<i>op0 (10)</i>	<i>S</i>	<i>I</i>	<i>R</i>	<i>RN</i>	<i>RD/RM</i>	Imediato/Endereço/ Offset
Bits	31:28	27:26	25	24	23	22:18	17:13	12:0

Este processador tem suporte para quatro tipos de modo de endereçamento à memória: imediato (para instruções *load*), direto, indireto por registrador e por deslocamento, descritos nas seções 3.5.1, 3.5.2, 3.5.3 e 3.5.4, respectivamente. O formato da Tabela 5 dá suporte para todos esses modos de endereçamento em uma única instrução.

O *opcode* é composto pelo *op0*, que para instruções de tipo M é 10, e 3 bits (*S*, *I* e *R*) que dizem respeito ao modo que a instrução acessa a memória. O bit *S* indica se a instrução é de tipo *load* (*S* = 0) ou *store* (*S* = 1). O bit *I*, quando setado, indica que a informação nos bits 12 a 0 será usada como imediato ou *offset*. Esse é o caso dos modos de endereçamento imediato e por deslocamento, respectivamente. No caso de uma instrução *load* com endereçamento direto, esses bits representam um endereço de memória e o bit *I* não é setado. Já o bit *R*, quando setado, indica um registrador base em *RN* (22:18), e é utilizado nos endereçamentos por deslocamento e indireto por registrador. Este tipo de instrução também está sujeita à execução condicional, e portanto também foi implementado o campo *cond* (bits 31:28).

Foram definidas duas instruções de tipo M, uma de *load* e outra de *store*, de mnemônicos LDR e STR, respectivamente. Essas instruções e demais detalhes a respeito de como acessam a memória estão descritos na Tabela 6.

Tabela 6 – Tipos, modos de endereçamento à memória, bits S, I e R e descrições das instruções do tipo M. Fonte: A autora (2025).

Tipo	Modo de Endereçamento	bits S, I e R	Descrição
<i>Load</i>	Imediato	010	Carrega imediato em RD
	Direto	000	Carrega palavra contida no endereço imediato da memória em RD
	Indireto por Registrador	001	Carrega palavra contida no endereço de memória que guarda RN em RD
<i>Store</i>	Deslocamento	011	Carrega palavra contida no endereço de memória de RN + offset em RD
	Direto	100	Guarda palavra contida em RM no endereço imediato de memória
	Indireto por Registrador	101	Guarda palavra contida em RM no endereço de memória contido em RN
	Deslocamento	111	Guarda palavra contida em RM no endereço de memória de RN + offset

Instruções do tipo M seguem os possíveis caminhos de dados ilustrados na Figura 8, baseados na estrutura da Figura 5.

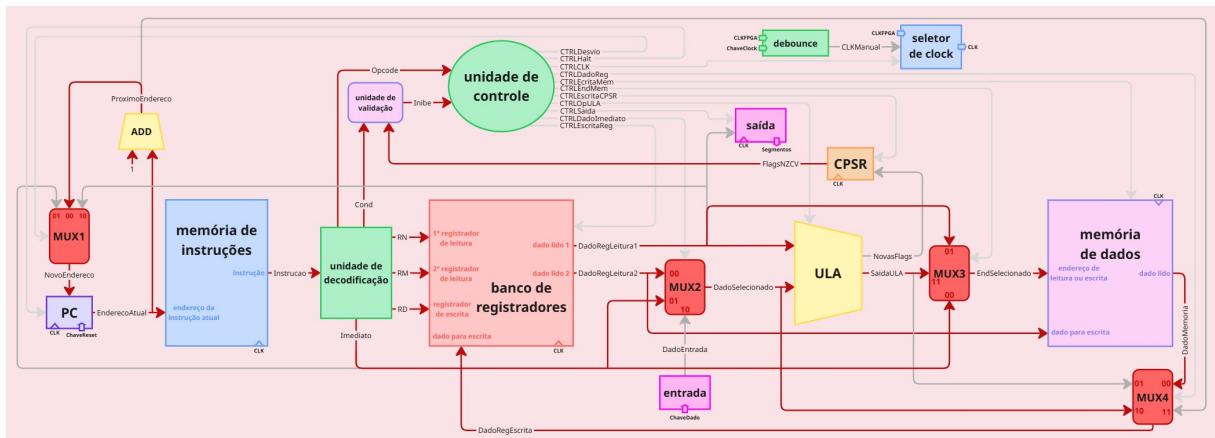


Figura 8 – Caminho de dados das instruções do tipo M. Fonte: A autora (2025).

4.3.4 Instruções Alternativas: Tipo A

São classificadas como alternativas instruções que não têm as mesmas funcionalidades descritas nas seções 4.3.1, 4.3.2 e 4.3.3. Estas serão referenciadas como instruções de tipo A, e estão codificadas de acordo com a Tabela 7.

Tabela 7 – Formato das instruções do tipo A. Fonte: A autora (2025).

Formato	<i>cond</i>	<i>op0</i>	<i>op1</i>	RN/RD	x
Bits	31:28	27:26	25:22	21:17	16:0

Instruções deste tipo não possuem uma função em comum. Portanto, seu formato foi projetado para ser flexível e expansível para possíveis futuras instruções.

Estas instruções também podem ser executadas condicionalmente, e assim também contam com um campo *cond*. O *opcode* é composto pelo *op0*, que para instruções do tipo A é 11, e pelo *op1*, que indica ao processador o que deve ser executado. Para instruções de E/S, como IN e OUT, os bits 21 a 17 guardam os endereços de registradores de escrita e leitura, respectivamente. A Tabela 8 descreve todas as instruções deste tipo, seus mnemônicos, *op1* e descrições.

Tabela 8 – Mnemônicos, *op1* e descrições das instruções do tipo A.
Fonte: A autora (2025).

Mnemônico	<i>op1</i>	Descrição
NOP	0000	Não faz nada
HLT	0001	Inibe a atualização do PC
IN	0010	Guarda valor de entrada do FPGA em RD
OUT	0011	Faz o display do valor de RN no FPGA

Instruções do tipo A seguem o caminho de dados ilustrado na Figura 9. No entanto, visto que podem assumir variadas funções, este pode variar dependendo da instrução.

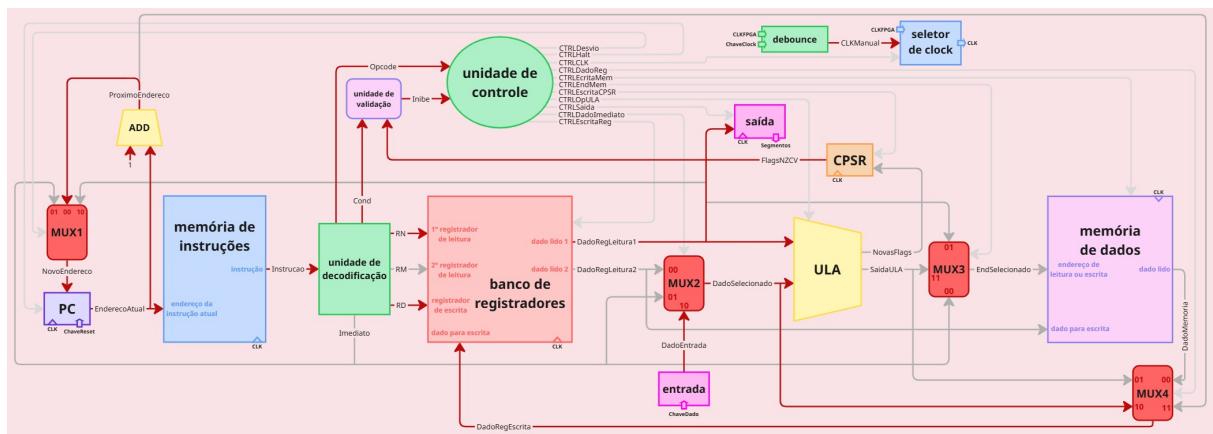


Figura 9 – Caminho de dados das instruções do tipo A. Fonte: A autora (2025).

4.4 Interface com FPGA

Conforme observado na Figura 5, existem alguns pontos de comunicação entre o processador e a placa FPGA. A Tabela 9 mostra as respectivas descrições e a pinagens desses elementos.

Tabela 9 – Descrições e pinagem dos pontos de comunicação do processador com o FPGA.

Elemento	Pinagem	Descrição
ChaveClock	SW17	Chave utilizada para atualização manual do <i>clock</i> no caso de uma instrução IN
Reset	SW16	Atualiza o valor guardado pelo PC para zero
ChaveDado	SW7-SW0	Chaves utilizadas para a entrada de dados no caso de uma instrução IN
Segmentos	HEX3-HEX0	<i>Displays</i> utilizados no caso de uma instrução OUT
CLKFPGA	<i>Clock</i> 50MHz	<i>Clock</i> de 50MHz do FPGA
LEDIN	LEDR0	LED que acende durante a execução de uma instrução IN

4.5 Implementação em Verilog

Cada unidade funcional do processador, conforme observado na Figura 5, foi implementada em Verilog. As seções 4.5.1 a 4.5.15 descrevem cada um desses elementos e suas implementações.

4.5.1 Program Counter (PC)

O PC, conforme descrito na seção 4.1, é um registrador especial de 32 bits que guarda o endereço da memória de instruções que contém a instrução sendo executada no momento atual. Essa informação é sempre atualizada com a borda de subida do *clock*, dado que o sinal de controle *CTRLHalt* esteja desativado. Caso contrário, a atualização do PC é inibida. Além disso, foi implementada uma função de *reset*, onde o valor guardado retorna a zero. Sua implementação em Verilog pode ser constatada na Figura 10.

```

Date: July 25, 2025          PC.v           Project: PC

1  module PC
2  (
3  // Entradas :: Endereço da Nova Instrução
4  input wire [31:0] NovoEndereco,
5
6  // Entradas :: Sinais de Controle
7  input wire CLK, CTRLHalt, Reset,
8
9  // Saídas :: Endereço da Instrução Atual
10 output reg [31:0] EnderecoAtual
11 );
12
13     initial
14     begin
15         EnderecoAtual = 32'b0;
16     end
17
18     always @ (posedge CLK)
19     begin
20         if (CTRLHalt == 1'b0)
21             begin
22                 EnderecoAtual <= NovoEndereco;
23             end
24         if (Reset == 1'b1)
25             begin
26                 EnderecoAtual <= 32'b0;
27             end
28     end
29
30 endmodule

```

Figura 10 – Código em Verilog da implementação do *Program Counter* (PC).

Fonte: A autora (2025).

4.5.2 Memória de Instruções

A memória de instruções, que guarda as instruções do programa sendo executado pelo processador, foi implementada com base num modelo de memória ROM do próprio *software* Quartus Prime, o *Single Port ROM*. Ela recebe *EnderecoAtual* como entrada, que vem do PC, e leva à saída *Instrucao* o dado guardado naquela posição de memória a cada borda de descida do *clock*. Inicialmente, por causa do menor tempo de compilação, essa memória foi declarada com 32 posições, cada uma com 32 bits. O módulo implementado pode ser visto na Figura 11.

```

Date: June 14, 2025           MemInstrucoes.v          Project: MemInstrucoes
1  module MemInstrucoes
2  #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
3  ( // Entradas :: Endereço da Instrução Atual
4    input [(ADDR_WIDTH-1):0] EnderecoAtual,
5
6    // Entradas :: Sinais de Controle
7    input CLK,
8
9    // Saídas :: Instrução Atual
10   output reg [(DATA_WIDTH-1):0] Instrucao
11 );
12
13   // Variáveis Auxiliares :: Memória ROM
14   reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];
15
16   initial
17     begin
18       $readmemb("MemInstrDados.txt", rom);
19     end
20
21   always @ (posedge CLK)
22     begin
23       Instrucao <= rom[EnderecoAtual];
24     end
25
26 endmodule

```

Figura 11 – Código em Verilog da implementação da memória de instruções.

Fonte: A autora (2025).

4.5.3 Unidade de Decodificação

A unidade de decodificação, conforme visto na seção 4.1, foi implementada com o objetivo de reconhecer o formato da instrução e distribuir ao resto do processador os campos adequados. Isso é feito por meio da verificação do campo op0 da instrução, representado pelos bits 27 e 26. Na seção 4.3, foi visto que para as instruções de tipo D, B, M e A, os campos op0 são 00, 01, 10 e 11, respectivamente. Assim, a Figura 12 mostra a implementação dessa unidade funcional em Verilog.

É possível notar que este bloco já tem a funcionalidade de extensão de sinal embutida. Isso acontece para alguns casos no *opcode* e nos campos de *offset* e imediato. No primeiro caso, a extensão é feita nos bits menos significativos, já que os MSB serão utilizados para o reconhecimento da instrução na unidade de controle. Além disso, ela ocorre porque, conforme visto na seção 4.3, os *opcodes* têm tamanho variável. Já no segundo caso, a extensão ocorre nos bits mais significativos a fim de não alterar a informação contida na instrução.

```

Date: June 14, 2025           UnidadeDecodificacao.v          Project: UnidadeDecodificacao
1  module UnidadeDecodificacao
2  (
3    // Entradas :: Instrução Atual
4    input wire [31:0] Instrucao,
5
6    // Saídas :: Cond & Opcode
7    output reg [3:0] Cond,
8    output reg [7:0] Opcode,
9
10   // Saídas :: Registradores
11   output reg [4:0] RN, RD, RM,
12
13   // Saídas :: Offset/Imediato
14   output reg [31:0] Imediato
15 );
16
17   // Parâmetros :: Tipos de Dados
18   localparam TipoD = 2'b00, TipoB = 2'b01, TipoM = 2'b10, TipoA = 2'b11;
19
20   // Outros Parâmetros
21   localparam RegVazio = 5'b0, LR = 5'b11111, BX = 2'b01, BL = 2'b10;
22
23   always @ (*)
24     begin
25       case (Instrucao[27:26])
26         TipoD:
27           begin
28             Cond    = Instrucao[31:28];
29             Opcode  = Instrucao[27:20];
30             RN      = Instrucao[19:15];
31             RD      = Instrucao[14:10];
32             RM      = Instrucao[9:5];
33             Imediato = {22'b0, Instrucao[9:0]};
34           end
35         TipoB:
36           begin
37             Cond    = Instrucao[31:28];
38             Opcode  = {Instrucao[27:24], 4'b0};
39             if (Instrucao[25:24] == BX)
40               begin
41                 RN    = Instrucao[23:19];
42                 RD    = RegVazio;
43               end
44             if (Instrucao[25:24] == BL)
45               begin
46                 RN    = RegVazio;
47                 RD    = LR;
48               end
49             RM    = RegVazio;
50             Imediato = {13'b0, Instrucao[18:0]};
51           end
52         TipoM:
53           begin
54             Cond    = Instrucao[31:28];
55             Opcode  = {Instrucao[27:22], 2'b0};
56             RN      = Instrucao[21:17];
57             RD      = Instrucao[16:12];
58             RM      = RegVazio;
59             Imediato = {20'b0, Instrucao[11:0]};
60           end
61         TipoA:
62           begin
63             Cond    = Instrucao[31:28];
64             Opcode  = {Instrucao[27:22], 2'b0};
65             RN      = RegVazio;
66             RD      = RegVazio;
67             RM      = RegVazio;
68             Imediato = 32'b0;
69           end
70       endcase
71     end
72
73 endmodule

```

Figura 12 – Código em Verilog da implementação da unidade de decodificação.

Fonte: A autora (2025).

4.5.4 Banco de Registradores

O banco de registradores, conforme descrito na seção 4.1, conta com 32 registradores de propósito geral de 32 bits cada. Este módulo realiza duas operações diferentes: leitura

e escrita em registradores.

Para a sua implementação em Verilog, vista na Figura 13, o módulo recebe como entrada dois registradores de leitura, RN e RM, e um de escrita, RD, os quais são codificados por 5 bits. Além disso, ele recebe um dado de 32 bits para a escrita em registrador, *DadoRegEscrita*, e dois sinais de controle, o próprio *clock* e *CTRLEscritaReg*, que vem da unidade de controle. As duas saídas do banco, *DadoRegLeitura1* e *DadoRegLeitura2*, representam os conteúdos dos registradores das entradas RN e RM, respectivamente.

O valor de RD só é atualizado com *DadoRegEscrita* quando o sinal *CTRLEscritaReg* está ativo e quando a borda de subida do *clock* é identificada. Assim, a escrita no banco de registradores é feita de forma sequencial. Já a leitura é feita de modo combinacional, e está sensível à mudança de qualquer entrada do módulo.

```
Date: June 14, 2025           BancoRegistradores.v          Project: BancoRegistradores
1  module BancoRegistradores
2  (
3    // Entradas :: Registradores de Leitura
4    input wire [4:0] RN, RM,
5
6    // Entradas :: Dados para Escrita
7    input wire [4:0] RD,
8    input wire [31:0] DadoRegEscrita,
9
10   // Entradas :: Sinais de Controle
11   input wire CTRLEscritaReg, CLK,
12
13   // Saídas :: Dados Lidos
14   output reg [31:0] DadoRegLeitura1, DadoRegLeitura2
15 );
16
17   // Variáveis Auxiliares :: Banco de Registradores
18   reg [31:0] Registradores[31:0];
19
20   always @ (*)
21     begin
22       DadoRegLeitura1 <= Registradores[RN];
23       DadoRegLeitura2 <= Registradores[RM];
24     end
25
26   always @ (posedge CLK)
27     begin
28       if (CTRLEscritaReg == 1'b1)
29         begin
30           Registradores[RD] <= DadoRegEscrita;
31         end
32     end
33
34 endmodule
```

Figura 13 – Código em Verilog da implementação do banco de registradores. Fonte: A autora (2025).

4.5.5 Unidade Lógica e Aritmética (ULA)

A ULA tem duas funções principais: realizar operações lógicas e aritméticas e, com base nestas, fazer o cálculo das novas *flags* NZCV do CPSR. Além dos dois dados de 32 bits, ela também recebe como entrada o sinal *CTRLOpULA* vindo da unidade de controle, que decide qual operação será realizada sobre os dados.

Baseado nas instruções da Tabela 2, a ULA realiza 9 operações distintas: soma, subtração (de duas maneiras diferentes, invertendo a ordem dos operandos), multiplicação,

divisão, *not*, *and*, *or* e *xor*. Assim, o sinal *CTRLOpULA* conta com 4 bits que descrevem 9 casos definidos, listados na Tabela 10. Quando *CTRLOpULA* não está definido, o sinal de controle não ativa nenhuma operação, caindo no caso *default*.

Tabela 10 – Operações codificadas pelo sinal de controle *CTRLOpULA*. Fonte: A autora (2025).

<i>CTRLOpULA</i>	Operação
0000	Soma
0001	Subtração 1
0010	Subtração 2
0011	Multiplicação
0100	Divisão
0101	<i>Not</i>
0110	<i>And</i>
0111	<i>Or</i>
1000	<i>Xor</i>
-	<i>Default</i>

Na implementação em Verilog, visto na Figura 14, a ULA recebe como entrada dois dados de 32 bits cada, *Dado1* e *Dado2*, e o sinal de controle *CTRLOpULA*, de 4 bits. O resultado de 32 bits da operação realizada na ULA, *SaidaULA*, e as novas *flags* de 4 bits, *FlagsCPSR*, calculadas em cima desse resultado são as saídas do módulo.

À *flag N* é atribuído o próprio MSB do resultado da ULA, que naturalmente indica o sinal do número, enquanto à *flag Z* é atribuído o valor da comparação do resultado com o número zero. Já o cálculo de *carry* e *overflow* é especialmente relevante para a soma e a subtração, e por isso foi implementado de maneira separada para essas operações.

Em especial para o cálculo da *flag C*, é utilizada a variável *AuxiliarCV*, que tem comprimento de 33 bits. Caso a operação resulte em *carry*, o MSB dessa variável será 1 e, caso contrário, será 0. Assim, à *flag C* é atribuído diretamente o MSB de *AuxiliarCV*. Já para o cálculo da *flag V*, são verificados os sinais dos operandos e do resultado, representados pelos seus respectivos MSB.

```

Date: June 15, 2025           ULA.v          Project: ULA
1  module ULA
2  (
3  // Entradas :: Dados
4  input wire [31:0] Dado1, Dado2,
5
6  // Entradas :: Sinais de Controle
7  input wire [3:0] CTRLOpULA,
8
9  // Saídas
10 output wire [31:0] SaídaULA,
11 output wire [3:0] NovasFlags
12 );
13
14  // Variáveis e Parâmetros Auxiliares
15 reg [32:0] AuxiliarCV;
16 reg [31:0] SaídaAuxiliar;
17 reg [1:0] FlagsAuxiliar;
18 localparam Zero = 32'b0;
19
20 // Parâmetros :: CTRLOpULA
21 localparam Soma = 4'b0000, Sub1 = 4'b0001, Sub2 = 4'b0010, Mul = 4'b0011,
22           Div = 4'b0100, Not = 4'b0101, And = 4'b0110, Or = 4'b0111, Xor =
23           4'b1000;
24
25 // Operações da ULA
26 always @ (CTRLOpULA or Dado1 or Dado2)
27 begin
28   FlagsAuxiliar[1:0] = 2'b0;
29   case(CTRLOpULA[3:0])
30     Soma: begin
31       AuxiliarCV = {1'b0, Dado1} + {1'b0, Dado2};
32       SaídaAuxiliar = AuxiliarCV[31:0];
33       FlagsAuxiliar[1] = AuxiliarCV[32];
34       FlagsAuxiliar[0] = (Dado1[31] == Dado2[31]) && (SaídaAuxiliar[31] != Dado1[
35         31]);
36     end
37     Sub1: begin
38       AuxiliarCV = {1'b0, Dado1} - {1'b0, Dado2};
39       SaídaAuxiliar = AuxiliarCV[31:0];
40       FlagsAuxiliar[1] = ~AuxiliarCV[32];
41       FlagsAuxiliar[0] = (Dado1[31] != Dado2[31]) && (SaídaAuxiliar[31] == Dado2[
42         31]);
43     end
44     Sub2: begin
45       AuxiliarCV = {1'b0, Dado2} - {1'b0, Dado1};
46       SaídaAuxiliar = AuxiliarCV[31:0];
47       FlagsAuxiliar[1] = ~AuxiliarCV[32];
48       FlagsAuxiliar[0] = (Dado1[31] != Dado2[31]) && (SaídaAuxiliar[31] == Dado1[
49         31]);
50     end
51     Mul: SaídaAuxiliar = Dado1 * Dado2;
52     Div: SaídaAuxiliar = Dado1 / Dado2;
53     Not: SaídaAuxiliar = ~Dado2;
54     And: SaídaAuxiliar = Dado1 & Dado2;
55     Or:  SaídaAuxiliar = Dado1 | Dado2;
56     Xor: SaídaAuxiliar = Dado1 ^ Dado2;
57     default: SaídaAuxiliar = Zero;
58   endcase
59 end
60
61 // Atribuição de valores às Saídas
62 assign SaídaULA = SaídaAuxiliar;
63 assign NovasFlags = {SaídaAuxiliar[31], (SaídaAuxiliar == Zero), FlagsAuxiliar[1:0]};
64
65 endmodule

```

Figura 14 – Código em Verilog da implementação da Unidade Lógica e Aritmética (ULA).
Fonte: A autora (2025).

4.5.6 Current Program Status Register (CPSR)

O CPSR, conforme visto na seção 4.1, é um registrador especial de 4 bits. Ele é essencial para a lógica da execução condicional de instruções pois guarda as quatro *flags* NZCV. Ele recebe como entrada as *flags* calculadas dentro da ULA, mas a informação só é atualizada com a borda de subida do *clock*, dado que o sinal *CPSRWrite*, que vem da unidade de controle, esteja ativo. A Figura 15 mostra o código em Verilog implementado

para o CPSR.

Date: June 14, 2025	CPSR.v	Project: CPSR
---------------------	--------	---------------

```

1  module CPSR
2  (
3    // Entradas :: Novas Flags Calculadas
4    input wire [3:0] NovasFlags,
5
6    // Entradas :: Sinais de Controle
7    input wire CTRLEscritaCPSR,
8    input wire CLK,
9
10   // Saídas :: Flags NZCV
11   output reg [3:0] FlagsNZCV
12 );
13
14   always @ (posedge CLK)
15   begin
16     if (CTRLEscritaCPSR == 1'b1)
17       begin
18         FlagsNZCV <= NovasFlags;
19       end
20     end
21
22 endmodule

```

Figura 15 – Código em Verilog da implementação do *Current Program Status Register* (CPSR). Fonte: A autora (2025).

4.5.7 Memória de Dados

A memória de dados, que é a memória principal do processador, foi implementada com base num modelo de memória RAM do próprio *software* Quartus Prime, o *Simple Dual Port RAM (Dual Clock)*. Inicialmente, por causa do menor tempo de compilação, essa memória foi declarada com 32 posições, cada uma com 32 bits. O módulo implementado foi alterado, e pode ser visto na Figura 16.

Assim como outros elementos de memória do processador, sua leitura funciona de maneira combinacional, enquanto sua escrita funciona de maneira sequencial. Assim, *DadoMemEscrita* só é escrito em *EndEscrita* na borda de subida do *clock*, quando *CTRLEscritaMem* está ativo. De maneira similar, *DadoMemoria* é atualizado toda vez que *EndLeitura* é atualizado.

```
Date: June 14, 2025           MemPrincipal.v          Project: MemPrincipal
1  module MemPrincipal
2  #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
3  (
4      // Entradas :: Endereços para Leitura e Escrita
5      input [(ADDR_WIDTH-1):0] EndLeitura, EndEscrita,
6
7      // Entradas :: Dado para Escrita
8      input [(DATA_WIDTH-1):0] DadoMemEscrita,
9
10     // Entradas :: Sinais de Controle
11     input CTRLEscritaMem, CLK,
12
13     // Saída :: Dado Lido
14     output reg [(DATA_WIDTH-1):0] DadoMemoria
15 );
16
17     // Variáveis Auxiliares :: Memória
18     reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
19
20     always @ (*)
21     begin
22         DadoMemoria = ram[EndLeitura];
23     end
24
25     always @ (posedge CLK)
26     begin
27         if (CTRLEscritaMem == 1'b1)
28             ram[EndEscrita] <= DadoMemEscrita;
29     end
30 endmodule
```

Figura 16 – Código em Verilog da implementação da memória de dados.

Fonte: A autora (2025).

4.5.8 Multiplexador (MUX)

É possível observar na Figura 5 que existem vários multiplexadores (MUX) presentes ao longo da estrutura interna do processador. Assim, foi implementada uma unidade multiplexadora única em Verilog, a qual pode ser observada na Figura 17, a fim de ser instanciada conforme necessário. Ela recebe como entrada 4 dados de 32 bits e uma entrada de seleção de 2 bits, com base nas quais determina a saída, também de 32 bits.

```
Date: June 14, 2025           MUX.v                Project: MUX
1  module MUX
2  (
3      // Entradas :: Dados
4      input wire [31:0] Dado1, Dado2, Dado3, Dado4,
5
6      // Entradas :: Entrada de Seleção
7      input wire [1:0] Selecao,
8
9      // Saídas :: Dado de Saída
10     output reg [31:0] DadoSaida
11 );
12
13     always @ (*)
14     begin
15         case (Selecao[1:0])
16             2'b00: DadoSaida <= Dado1;
17             2'b01: DadoSaida <= Dado2;
18             2'b10: DadoSaida <= Dado3;
19             2'b11: DadoSaida <= Dado4;
20         endcase
21     end
22
23 endmodule
```

Figura 17 – Código em Verilog da implementação do multiplexador (MUX).

Fonte: A autora (2025).

4.5.9 Adder

Análogo ao MUX, é possível observar uma instância de somador, ou *adder*, na estrutura do processador ilustrada na Figura 5. Assim, foi também projetado um módulo que soma duas entradas, cuja implementação pode ser observada na Figura 18.

Date: June 14, 2025	Adder.v	Project: Adder
<pre> 1 module Adder 2 (3 // Entradas :: Dados 4 input wire [31:0] Dado1, Dado2, 5 6 // Saída :: Soma 7 output reg [31:0] Soma 8); 9 10 always @ (*) 11 begin 12 Soma <= Dado1 + Dado2; 13 end 14 15 endmodule </pre>		

Figura 18 – Código em Verilog da implementação do somador ou *adder*.

Fonte: A autora (2025).

4.5.10 Unidade de Validação

A unidade de validação é responsável por comparar o campo *cond* às *flags NZCV* do CPSR, e assim gerar um sinal que decide pela permissão ou inibição da execução da instrução atual. Essa comparação é feita com base no que é estipulado na Figura 2. Sua implementação em Verilog pode ser observada na Figura 19.

```

Date: June 15, 2025                                         UnidadeValidacao.v                                         Project: UnidadeValidacao
1  module Unidadevalidacao
2  (
3    // Entradas
4    input wire [3:0] Cond, FlagsNZCV,
5    // Saídas
6    output reg Inibe
7  );
8
9
10   localparam EQ = 4'b0000, NE = 4'b0001, CS = 4'b0010, CC = 4'b0011, MI = 4'b0100,
11     PL = 4'b0101, VS = 4'b0110, VC = 4'b0111, HI = 4'b1000, LS = 4'b1001,
12     GE = 4'b1010, LT = 4'b1011, GT = 4'b1100, LE = 4'b1101;
13   localparam N = 1'd3, Z = 1'd2, C = 1'd1, V = 1'd0;
14
15   always @ (*)
16   begin
17     case(Cond[3:0])
18       EQ: begin
19         if (FlagsNZCV[Z] == 1'b1)
20           begin Inibe <= 1'b0; end
21         else
22           begin Inibe <= 1'b1; end
23       end
24       NE: begin
25         if (FlagsNZCV[Z] == 1'b0)
26           begin Inibe <= 1'b0; end
27         else
28           begin Inibe <= 1'b1; end
29       end
30       CS: begin
31         if (FlagsNZCV[C] == 1'b1)
32           begin Inibe <= 1'b0; end
33         else
34           begin Inibe <= 1'b1; end
35       end
36       CC: begin
37         if (FlagsNZCV[C] == 1'b0)
38           begin Inibe <= 1'b0; end
39         else
40           begin Inibe <= 1'b1; end
41       end
42       MI: begin
43         if (FlagsNZCV[N] == 1'b1)
44           begin Inibe <= 1'b0; end
45         else
46           begin Inibe <= 1'b1; end
47       end
48       PL: begin
49         if (FlagsNZCV[N] == 1'b0)
50           begin Inibe <= 1'b0; end
51         else
52           begin Inibe <= 1'b1; end
53       end
54
55   VS: begin
56     if (FlagsNZCV[V] == 1'b1)
57       begin Inibe <= 1'b0; end
58     else
59       begin Inibe <= 1'b1; end
60   end
61   VC: begin
62     if (FlagsNZCV[V] == 1'b0)
63       begin Inibe <= 1'b0; end
64     else
65       begin Inibe <= 1'b1; end
66   end
67   HI: begin
68     if (FlagsNZCV[C] == 1'b1 && FlagsNZCV[Z] == 1'b0)
69       begin Inibe <= 1'b0; end
70     else
71       begin Inibe <= 1'b1; end
72   end
73   LS: begin
74     if (FlagsNZCV[C] == 1'b0 || FlagsNZCV[Z] == 1'b1)
75       begin Inibe <= 1'b0; end
76     else
77       begin Inibe <= 1'b1; end
78   end
79   GE: begin
80     if (FlagsNZCV[N] == FlagsNZCV[V])
81       begin Inibe <= 1'b0; end
82     else
83       begin Inibe <= 1'b1; end
84   end
85   LT: begin
86     if (FlagsNZCV[N] != FlagsNZCV[V])
87       begin Inibe <= 1'b0; end
88     else
89       begin Inibe <= 1'b1; end
90   end
91   GT: begin
92     if (FlagsNZCV[Z] == 1'b0 && FlagsNZCV[N] == FlagsNZCV[V])
93       begin Inibe <= 1'b0; end
94     else
95       begin Inibe <= 1'b1; end
96   end
97   LE: begin
98     if (FlagsNZCV[Z] == 1'b1 || FlagsNZCV[N] != FlagsNZCV[V])
99       begin Inibe <= 1'b0; end
100    else
101      begin Inibe <= 1'b1; end
102    default: Inibe <= 1'b0;
103  endcase
104 end
105
106 endmodule

```

(a) Linhas 1 a 53.

```

Date: June 15, 2025                                         UnidadeValidacao.v                                         Project: UnidadeValidacao
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106

```

(b) Linhas 54 a 106.

Figura 19 – Código em Verilog da implementação da unidade de validação. Fonte: A autora (2025).

4.5.11 Integração Memória de Instruções e Unidade de Decodificação

Dados os módulos desenvolvidos para a memória de instruções e unidade de decodificação, vistos nas Figuras 11 e 12, respectivamente, foi implementado o módulo da sua integração, conforme previsto na Figura 5. O código em Verilog implementado para isso pode ser observado na Figura 20.

```
Date: June 14, 2025           IntegracaoMemInstrUnidDecod.v       Project: IntegracaoMemInstrUnidDecod
1  module IntegracaoMemInstrUnidDecod
2  (
3    // Entradas :: Endereço da Instrução Atual
4    input wire [31:0] EnderecoAtual,
5
6    // Entradas :: Clock
7    input wire CLK,
8
9    // Saídas :: Cond & Opcode
10   output wire [3:0] Cond,
11   output wire [7:0] opcode,
12
13   // Saídas :: Registradores
14   output wire [4:0] RN, RD, RM,
15
16   // Saídas :: Offset/Imediato
17   output wire [31:0] Imediato
18 );
19
20   wire [31:0] Instrucao;
21
22   MemInstrucoes MemInstrucoes  (.EnderecoAtual(EnderecoAtual),
23                                 .CLK(CLK),
24                                 .Instrucao(Instrucao)  );
25
26   UnidadeDecodificacao UnidadeDecodificacao (.Instrucao(Instrucao),
27                                               .Cond(Cond),
28                                               .Opcode(opcode),
29                                               .RN(RN),
30                                               .RD(RD),
31                                               .RM(RM),
32                                               .Imediato(Imediato)  );
33
34 endmodule
```

Figura 20 – Código em Verilog da implementação do módulo de integração entre memória de instruções e unidade de decodificação. Fonte: A autora (2025).

4.5.12 Integração Banco de Registradores e ULA

As implementações do banco de registradores e da ULA, vistos nas Figuras 13 e 14, respectivamente, foram integrados em um único módulo. O código em Verilog dessa integração pode ser observado na Figura 21.

```
Date: June 05, 2025           IntegracaoBancoULA.v          Project: IntegracaoBancoULA
1  module IntegracaoBancoULA ( // Entradas :: Registradores de Leitura
2    input wire [4:0] RegLeitura1,
3    input wire [4:0] RegLeitura2,
4
5    // Entradas :: Dados para Escrita
6    input wire [4:0] RegEscrita,
7    input wire [31:0] DadoEscrita,
8
9    // Entradas :: Sinais de Controle BR
10   input wire RegWrite,
11   input wire Clock,
12
13   // Entradas :: Sinais de Controle ULA
14   input wire [3:0] OpULA,
15
16   // Saídas
17   output wire [31:0] SaídaULA,
18   output wire [3:0] FlagsCPSR  );
19
20  // Variáveis Auxiliares
21  wire [31:0] DadoULA1;
22  wire [31:0] DadoULA2;
23
24  BancoRegistradores BancoRegistradores (
25    .RegLeitura1(RegLeitura1),
26    .RegLeitura2(RegLeitura2),
27    .RegEscrita(RegEscrita),
28    .DadoEscrita(DadoEscrita),
29    .RegWrite(RegWrite),
30    .Clock(Clock),
31    .DadoLido1(DadoULA1),
32    .DadoLido2(DadoULA2)      );
33
34  ULA ULA (   .Dado1(DadoULA1),
35    .Dado2(DadoULA2),
36    .OpULA(OpULA),
37    .SaídaULA(SaídaULA),
38    .FlagsCPSR(FlagsCPSR)  );
39
40 endmodule
```

Figura 21 – Código em Verilog da implementação do módulo de integração entre banco de registradores e Unidade Lógica e Aritmética (ULA). Fonte: A autora (2025).

4.5.13 Entrada e Saída (E/S)

O módulo de entrada lê o dado das 8 chaves definidas na Tabela 9 e faz sua respectiva extensão de sinal. Sua saída é então salva no registrador especificado no campo RD de uma instrução IN. A implementação desse módulo em Verilog pode ser observada na Figura 22.

```
Date: July 25, 2025           Entrada.v          Project: Entrada
1  module Entrada
2  (
3    // Entradas :: Chaves FPGA
4    input wire [7:0] Chaves,
5
6    // Saídas :: Dado
7    output reg [31:0] Dado
8  );
9
10  always @ (*)
11    begin
12      Dado <= {24'b0, Chaves[7:0]};
13    end
14
15 endmodule
```

Figura 22 – Código em Verilog da implementação do módulo de entrada. Fonte: A autora (2025).

O módulo de saída recebe o dado de um registrador como entrada e faz o seu *display* no FPGA, conforme estabelecido na Tabela 9. O valor do qual se faz o *display* só é atualizado com a borda de subida do *clock*, dado que o sinal de controle *CTRLSaída* esteja ativo. Sua implementação em Verilog pode ser observada na Figura 23.

```
Date: July 25, 2025           Saída.v           Project: Saída
1  module Saída
2  (
3    // Entradas :: Dado
4    input wire [31:0] DadoEntrada,
5
6    // Entradas :: Sinais de Controle
7    input wire CTRLSaída, CLK,
8
9    // Saídas :: LEDs dos Displays de 7 Segmentos
10   output wire [27:0] Segmentos
11 );
12
13   wire [3:0] MilharBCD, CentenaBCD, DezenaBCD, UnidadeBCD;
14   reg [31:0] DadoSaída;
15
16   initial
17   begin
18     DadoSaída = 32'd0;
19   end
20
21   always @ (posedge CLK)
22   begin
23     if (CTRLSaída == 1'b1)
24       begin
25         DadoSaída <= DadoEntrada;
26       end
27     end
28
29   BinarioBCD BinarioBCD  (
30     .Dado(DadoSaída[13:0]),
31     .Milhar(MilharBCD),
32     .Centena(CentenaBCD),
33     .Dezena(DezenaBCD),
34     .Unidade(UnidadeBCD)
35   );
36
37   BCD7Seg Milhar      (
38     .BCD(MilharBCD),
39     .Segmentos(Segmentos[27:21])
40   );
41
42   BCD7Seg Centena    (
43     .BCD(CentenaBCD),
44     .Segmentos(Segmentos[20:14])
45   );
46
47   BCD7Seg Dezena    (
48     .BCD(DezenaBCD),
49     .Segmentos(Segmentos[13:7])
50   );
51
52   BCD7Seg Unidade   (
53     .BCD(UnidadeBCD),
54     .Segmentos(Segmentos[6:0])
55   );
56
57 endmodule
```

Figura 23 – Código em Verilog da implementação do módulo de saída. Fonte: A autora (2025).

Para a implementação desse módulo, foram precisos também módulos auxiliares de conversão de número binário para código BCD e de decodificação de BCD para *display* de 7 segmentos. Esses códigos em Verilog podem ser observados na Figura 24.

```
Date: July 25, 2025 BinarioBCD.v Project: BinarioBCD
1 module BinarioBCD
2 (
3 // Entradas :: Dado de 8 Bits
4 input wire [13:0] Dado,
5
6 // Saídas :: BCD
7 output reg [3:0] Milhar, Centena, Dezena, Unidade
8 );
9
10 always @ (*)
11 begin
12 Milhar <= Dado / 1000;
13 Centena <= (Dado % 1000) / 100;
14 Dezena <= ((Dado % 1000) % 100) / 10;
15 Unidade <= ((Dado % 1000) % 100) % 10;
16 end
17
18 endmodule
```

(a) Conversor de número binário para código BCD.

```
Date: July 25, 2025 BCD7Seg.v Project: BCD7Seg
1 module BCD7Seg
2 (
3 // Entradas :: BCD
4 input wire [3:0] BCD,
5
6 // Saídas :: LEDs do Display de 7 Segmentos
7 output reg [6:0] Segmentos
8 );
9
10 always @ (*)
11 begin
12 case(BCD[3:0])
13 4'b0000: Segmentos = 7'b1000000;
14 4'b0001: Segmentos = 7'b1111001;
15 4'b0010: Segmentos = 7'b0100100;
16 4'b0011: Segmentos = 7'b0111000;
17 4'b0100: Segmentos = 7'b0011001;
18 4'b0101: Segmentos = 7'b00110010;
19 4'b0110: Segmentos = 7'b00000010;
20 4'b0111: Segmentos = 7'b1111000;
21 4'b1000: Segmentos = 7'b00000000;
22 4'b1001: Segmentos = 7'b00110000;
23 default: Segmentos = 7'b1111111;
24 endcase
25 end
26
27 endmodule
```

(b) Decodificador BCD para *display* de 7 segmentos.

Figura 24 – Códigos em Verilog das implementações dos módulos auxiliares à saída.

Fonte: A autora (2025).

4.5.14 Unidade de *Debouncing* e Seletor de *Clock*

A implementação do módulo de entrada requer que todo o processamento sendo realizado seja pausado até que um dado seja entrado. Assim, o seletor de *clock* foi pensado como forma de trocar o *clock* de operação do processador para uma entrada manual. Ele também tem a função embutida de divisor de frequência. Sua implementação pode ser observada na Figura 25.

```

Date: July 25, 2025           SeletorClock.v          Project: SeletorClock
1  module SeletorClock
2  (
3  // Entradas :: Alternativas de clock
4  input wire CLKFPGA, CLKManual,
5
6  // Entradas :: Sinais de Controle
7  input wire CTRLCLK,
8
9  // Saída :: Clock
10 output reg CLK
11 );
12
13     reg [23:0] Contador;
14     reg CLKDividido;
15
16     initial
17     begin
18         Contador = 24'b0;
19         CLKDividido = 1'b0;
20     end
21
22     always @ (posedge CLKFPGA)
23     begin
24         if (CTRLCLK == 1'b1)
25             begin
26                 Contador <= 24'b0;
27                 CLKDividido <= 1'b0;
28             end
29         else
30             begin
31                 if (Contador == 24'd10000000)
32                     begin
33                         Contador <= 24'b0;
34                         CLKDividido <= ~CLKDividido;
35                     end
36                 else
37                     begin
38                         Contador <= Contador + 1'b1;
39                     end
40             end
41     end
42
43     always @ (*)
44     begin
45         if (CTRLCLK == 1'b0)
46             begin
47                 CLK <= CLKDividido;
48             end
49         else
50             begin
51                 CLK <= CLKManual;
52             end
53     end
54
55 endmodule

```

Figura 25 – Código em Verilog da implementação do seletor de *clock*. Fonte: A autora (2025).

O uso de uma chave da placa de FPGA como sinal de *clock* manual está sujeito a distorções mecânicas, o que foi evitado implementando-se uma unidade de *debouncing*. O código em Verilog desse módulo pode ser observado na Figura 26.

```

Date: July 25, 2025          Debounce.v           Project: Debounce
1  module Debounce
2  (
3    input wire CLKFPGA, ChaveClock,
4    output reg CLKManual
5  );
6
7    reg [23:0] Contador;
8
9    initial
10   begin
11     Contador = 24'b0;
12   end
13
14  always @ (posedge CLKFPGA)
15  begin
16    if (ChaveClock == 1'b1)
17      begin
18        if (Contador == 24'd500000)
19          begin
20            CLKManual <= 1'b1;
21            Contador <= 24'b0;
22          end
23        else
24          begin
25            Contador <= Contador + 1'b1;
26          end
27      end
28    else
29      begin
30        CLKManual <= 1'b0;
31        Contador <= 24'b0;
32      end
33    end
34  endmodule
35

```

Figura 26 – Código em Verilog da implementação da unidade de *debouncing*. Fonte: A autora (2025).

4.5.15 Integração Completa

A integração completa de todas as unidades funcionais do processador, conforme esquematizado na Figura 5, pode ser observada nas Figuras 27 e 28.

```

Date: July 25, 2025           Integracao.v          Project: Integracao
1  module Integracao
2  (
3    input wire [7:0] ChaveDado,
4    input wire CLKFPGA, ChaveClock, ChaveReset,
5    output wire [27:0] Segmentos,
6    output wire LEDIN
7  );
8    wire [31:0] NovoEndereco, EnderecoAtual, ProximoEndereco, Instrucao, EndSelected,
9    Immediato;
10   wire [31:0] DadoEntrada, DadoRegLeitura1, DadoRegLeitura2, DadoSelected,
11     DadoRegEscrita, SaídaULA, DadoMemoria;
12   wire [7:0] Opcode;
13   wire [4:0] RN, RM, RD;
14   wire [3:0] Cond, NovasFlags, FlagsNZCV;
15   wire [3:0] Inibe;
16   wire [3:0] CTRLOpULA;
17   wire [1:0] CTRLDesvio, CTRLDadosImmediato, CTRLEndMem, CTRLDadosReg;
18   wire CTRLEscritaReg, CTRLEscritaCPSR, CTRLEscritaMem, CTRLHalt, CTRLSaída, CTRLCLK;
19   wire CLK, CLKManual;
20
21   localparam Zero = 32'b0, Um = 32'b00000000000000000000000000000000;
22
23   PC PC (.CTRLHalt(CTRLHalt),
24           .CLK(CLK),
25           .Reset(ChaveReset),
26           .NovoEndereco(NovoEndereco),
27           .EnderecoAtual(EnderecoAtual)
28 );
29
30   Adder ADD (.Dado1(EnderecoAtual),
31               .Dado2(Um),
32               .Soma(ProximoEndereco)
33 );
34
35   MUX MUX1 (.Dado1(ProximoEndereco),
36             .Dado2(Immediato),
37             .Dado3(DadoRegLeitura1),
38             .Dado4(zero),
39             .Seleção(CTRLDesvio),
40             .DadosSaida(NovoEndereco)
41 );
42
43   MemInstruções MemInstruções (.EnderecoAtual(EnderecoAtual),
44                               .CLK(CLK),
45                               .Instrucao(Instrucao)
46 );
47
48   UnidadeDecodificação UnidadeDecodificação (.Instrucao(Instrucao),
49                                               .Cond(Cond),
50                                               .Opcode(Opcode),
51                                               .RN(RN),
52                                               .RM(RM),
53                                               .RD(RD),
54                                               .Immediato(Immediato)
55 );
56
57   BancoRegistradores BancoRegistradores (.RN(RN),
58                                         .RM(RM),
59                                         .RD(RD),
60                                         .DadosRegEscrita(DadoRegEscrita),
61                                         .CTRLEscritaReg(CTRLEscritaReg),
62                                         .CLK(CLK),
63                                         .DadosRegLeitura1(DadoRegLeitura1),
64                                         .DadosRegLeitura2(DadoRegLeitura2)
65 );
66
67   Entrada Entrada (.Chaves(ChaveDado),
68                     .Dado(DadoEntrada)
69 );
70
71   MUX MUX2 (.Dado1(DadoRegLeitura2),
72             .Dado2(Immediato),
73             .Dado3(DadoEntrada),
74             .Dado4(zero),
75             .Seleção(CTRLDadosImmediato),
76             .DadosSaida(DadoSelected)
77 );

```

Figura 27 – Linhas 1 a 75 do código em Verilog da integração completa do processador.
Fonte: A autora (2025).

```

Date: July 25, 2025           Integracao.v          Project: Integracao
76
77     Saida Saida ( .DadoEntrada(DadoRegLeitura1),
78             .CTRLSaida(CTRLSaida),
79             .CLK(CLK),
80             .Segmentos(Segmentos)
81         );
82
83     ULA ULA ( .Dado1(DadoRegLeitura1),
84             .Dado2(DadoSelecionado),
85             .CTRLOpULA(CTRLOpULA),
86             .SaidaULA(SaidaULA),
87             .NovasFlags(NovasFlags)
88         );
89
90     CPSR CPSR ( .NovasFlags(NovasFlags),
91             .CTRLEscritaCPSR(CTRLEscritaCPSR),
92             .CLK(CLK),
93             .FlagsNZCV(FlagsNZCV)
94         );
95
96     MUX MUX3 ( .Dado1(Imediato),
97             .Dado2(DadoRegLeitura1),
98             .Dado3(Zero),
99             .Dado4(SaidaULA),
100            .Selecao(CTRLEndMem),
101            .DadoSaida(EndSelecionado)
102        );
103
104    MemPrincipal MemPrincipal ( .EndLeitura(EndSelecionado),
105             .EndEscrita(EndSelecionado),
106             .DadoMemEscrita(DadoRegLeitura2),
107             .CTRLEscritaMem(CTRLEscritaMem),
108             .CLK(CLK),
109             .DadoMemoria(DadoMemoria)
110         );
111
112    MUX MUX4 ( .Dado1(DadoMemoria),
113             .Dado2(SaidaULA),
114             .Dado3(DadoSelecionado),
115             .Dado4(ProximoEndereco),
116             .Selecao(CTRLDadoReg),
117             .DadoSaida(DadoRegEscrita)
118         );
119
120    Unidadevalidacao Unidadevalidacao ( .Cond(Cond),
121             .FlagsNZCV(FlagsNZCV),
122             .Inibe(Inibe)
123         );
124
125    UnidadeControle UnidadeControle ( .Opcode(Opcode),
126             .Inibe(Inibe),
127             .CTRLDesvio(CTRLDesvio),
128             .CTRLDadoImediato(CTRLDadoImediato),
129             .CTRLEndMem(CTRLEndMem),
130             .CTRLDadoReg(CTRLDadoReg),
131             .CTRLEscritaReg(CTRLEscritaReg),
132             .CTRLEscritaCPSR(CTRLEscritaCPSR),
133             .CTRLEscritaMem(CTRLEscritaMem),
134             .CTRLOpULA(CTRLOpULA),
135             .CTRLHalt(CTRLHalt),
136             .CTRLSaida(CTRLSaida),
137             .CTRLCLK(CTRLCLK),
138             .LEDIN(LEDIN)
139         );
140
141    SeletorClock SeletorClock ( .CLKFPGA(CLKFPGA),
142             .CLKManual(CLKManual),
143             .CTRLCLK(CTRLCLK),
144             .CLK(CLK)
145         );
146
147    Debounce Debounce ( .CLKFPGA(CLKFPGA),
148             .ChaveClock(ChaveClock),
149             .CLKManual(CLKManual)
150         );
151
152 endmodule

```

Figura 28 – Linhas 76 a 152 do código em Verilog da integração completa do processador.

Fonte: A autora (2025).

5 Resultados e Discussão

Até aqui, foi feita uma descrição completa do projeto a fim de serem compreendidos todos os aspectos do processador. A seção 4.1 descreve todas as características da arquitetura adotada, incluindo alguns ajustes necessários, e a sua organização estrutural interna. A seção 4.2 detalha como o processador trata da execução condicional de instruções, que é característica da arquitetura ARM, e as seções 4.3 e 4.4 se aprofundam no conjunto de instruções e todas as suas particularidades e no sistema de interface com a placa FPGA, respectivamente. A fim de se comprovar o funcionamento das unidades funcionais do processador implementadas em Verilog, apresentadas na seção 4.5, foram simuladas formas de onda no *software* Quartus Prime. Estas são desenvolvidas nas seções 5.1 a 5.14.

5.1 Program Counter (PC)

O código da Figura 10 foi compilado e usado para a simulação de formas de onda correspondente às entradas e saídas do PC, visto na Figura 29. Nota-se que a saída *EnderecoAtual* é atualizada com o valor da entrada *NovoEndereco* a cada borda de subida do *clock* entre 0 e 160ns, quando o sinal *CTRLHalt* está desativado. A partir de 160ns, a saída do módulo para de ser atualizada, e a partir de 200ns, quando *Reset* está ativado, o valor de PC volta ao 0. Assim, comprova-se o funcionamento desta unidade do processador.

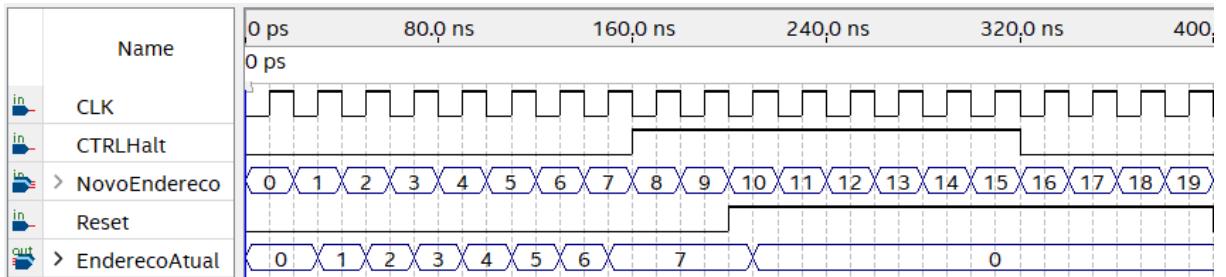


Figura 29 – Simulação de formas de onda do *Program Counter* (PC) gerada no *software* Quartus Prime.

5.2 Memória de Instruções

A memória de instruções é carregada por meio da leitura de um arquivo txt contendo diferentes informações em binário, conforme pode ser observado no código em Verilog implementado na Figura 11. Para fins de teste desta unidade do processador, foram carregadas inicialmente 8 instruções diferentes, as quais estão listadas na Tabela 11.

Tabela 11 – Instruções carregadas na memória de instruções, posições na memória e valores correspondentes em decimal. Fonte: A autora (2025).

Instrução em Binário	Posição na Memória	Decimal
01100000110000110000010100100000	0	1623393568
100100100010001000111101001100	1	2451644236
00100101001100000000000000000000	2	623902720
01110110111100001110011010011100	3	1995499164
00001001110000100010000011010001	4	163717329
01001010010001100101000000000000	5	1246121984
11011101100000000000000000000001	6	3716153347
11101100010000000000000000000000	7	3963617280

Assim, é possível observar na Figura 30, que mostra a simulação de formas de onda da memória de instruções, que a saída *Instrucao* é atualizada com o conteúdo da memória em *EnderecoAtual* a cada borda de descida do *clock*. Vale destacar que os valores da saída, que foram transformados em números decimais sem sinal para melhor interpretação dos dados, correspondem aos valores das instruções da Tabela 11 em decimal. Dessa maneira, fica evidente o funcionamento adequado da memória de instruções.

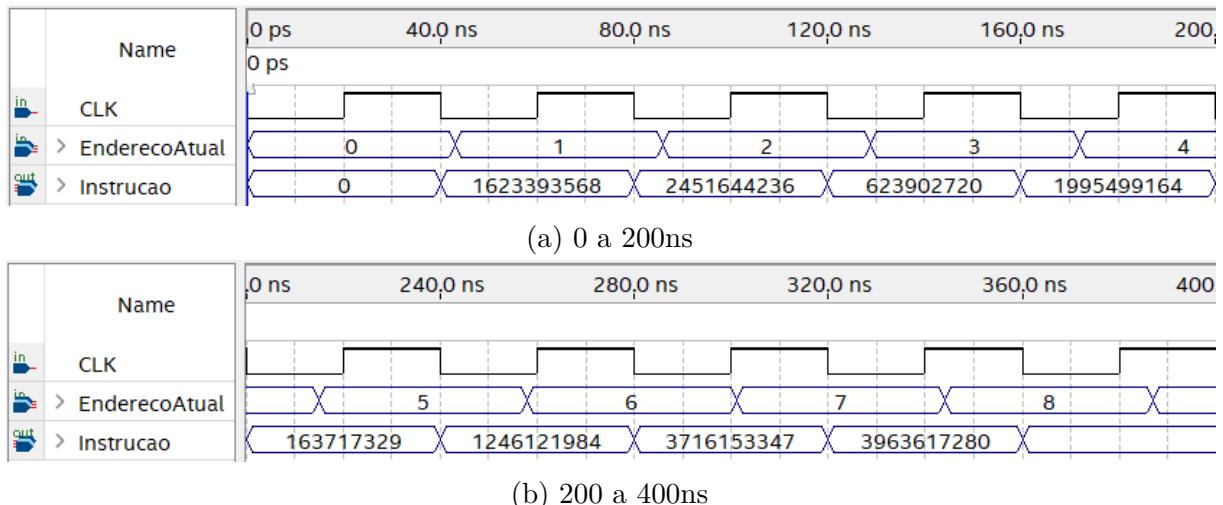


Figura 30 – Simulação de formas de onda da memória de instruções gerada no software Quartus Prime.

5.3 Unidade de Decodificação

A integração da memória de instruções com a unidade de decodificação, conforme descrito na seção 4.5.11, foi feita inicialmente para testar a funcionalidade da última. Assim, a Figura 31 apresenta a simulação de formas de onda do módulo dessa integração, que foi feita com base nas intruções da Tabela 11.

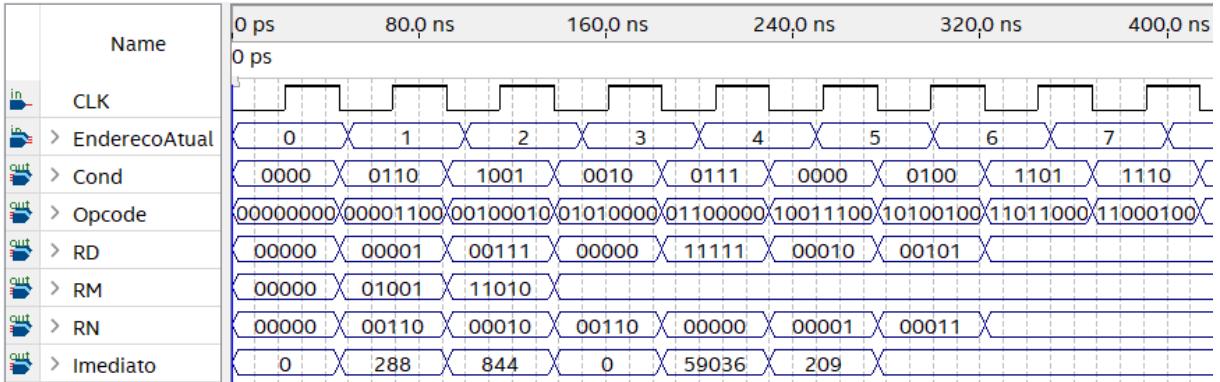


Figura 31 – Simulação de formas de onda do módulo de integração da memória de instruções e unidade de decodificação gerada no *software* Quartus Prime.

É possível notar que o valor das saídas do módulo são atualizadas a cada borda de descida do *clock*, com base no valor da entrada *EnderecoAtual*. Para as instruções que estão nos endereços de memória 0 e 1, que são instruções do tipo D, observa-se que a distribuição dos campos ocorre de maneira adequada, de acordo com o que é atribuído na implementação em Verilog. O campo *Imediato*, em específico, tem seu valor estendido de maneira que não muda seu valor, conforme descrito na seção 4.5.3.

As próximas duas instruções, que são do tipo B, carregadas nas posições 2 e 3 da memória, também apresentam comportamento correto. A primeira é uma instrução BX, enquanto a segunda é uma instrução BL. Aqui, é interessante destacar a atribuição dos campos RN e RD, que ocorre conforme visto em na Figura 12. Para a instrução BL, independente do valor contido no campo do registrador, o valor atribuído a RD é o valor de LR, que conforme explicitado na seção 4.1, é o registrador R31.

Para as instruções nos endereços 4 e 5, que, de acordo com a Tabela 11, são do tipo M, o módulo implementado distribui corretamente as informações que carregam. Em específico para a instrução do endereço 5, vale notar que o bit I está setado, o que indica que o campo *Imediato* não será usado para a execução da instrução. As instruções do tipo A, que estão nos endereços 6 e 7 da memória, também apresentam distribuição correta, com os campos RN, RD, RM e *Imediato* recebendo valores nulos.

5.4 Banco de Registradores

A Figura 32 mostra a simulação de formas de onda obtida a partir do código da Figura 13, módulo que implementa o banco de registradores em Verilog. É possível observar que entre 0 e 160ns, quando o sinal de controle *CTRLEscritaReg* está ativo, dados são escritos nos registradores R0 a R15 a cada borda de subida do *clock*. Esses dados são levados à saída *DadoRegLeitura1* entre 0 e 320ns conforme a mudança da entrada RN. Esses registradores são sobreescritos entre 320 e 480ns, o que pode ser observado na saída

DadoRegLeitura2 entre 320 e 640ns conforme a variação da entrada RM.

Os registradores R16 a R31 estão inicialmente vazios, conforme mostra *DadoRegLeitura2* com a variação da entrada RM entre 0 e 320ns. Já entre 160 e 320ns, quando o sinal de controle *CTRLEscritaReg* está desativado, esses registradores não são sobreescritos. Assim, entre 320 e 640ns, percebe-se novamente que estão vazios, dessa vez na saída *DadoRegLeitura1* com a variação da entrada RN. Dessa maneira, conclui-se que o banco de registradores implementado funciona de acordo com o esperado.

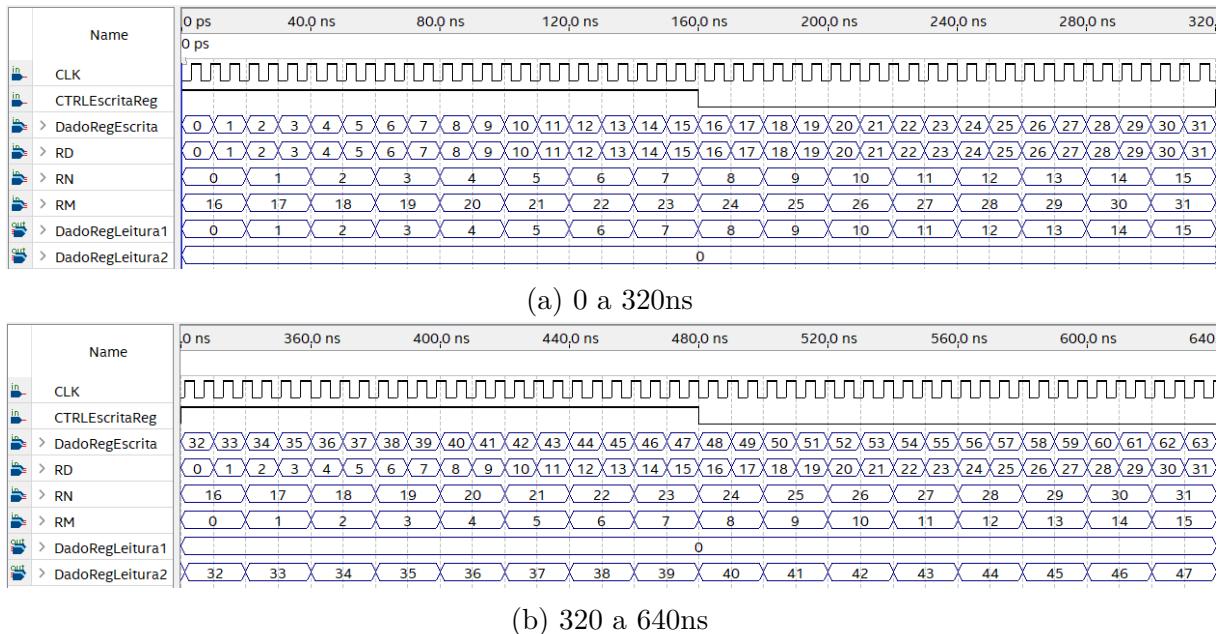


Figura 32 – Simulação de formas de onda do banco de registradores gerada no *software Quartus Prime*.

5.5 Unidade Lógica e Aritmética (ULA)

A Figura 33 mostra a simulação de formas de onda correspondente ao módulo da ULA, visto na Figura 14. Nos intervalos de aproximadamente 0 a 135ns e 240 a 375ns, fica evidente o funcionamento correto das operações da ULA, desde a soma até o *xor*. É possível perceber também que, para operações em que o resultado *SaidaULA* é negativo, a *flag N* está setada. Esse é o caso entre aproximadamente 30 e 45ns, para -1, assim como entre 270 e 285ns, para -5. Já quando o resultado da operação é 0, percebe-se que a *flag Z* é setada, o que é o caso entre 0 a 30ns, 45 a 60ns, 135 a 240ns e 375 e 480ns. Em especial, nestes últimos dois intervalos, o resultado é 0 pois o sinal *CTRLOpULA* não está definido, conforme visto na Tabela 10, o que cai no caso *default* e atribui a *SaidaULA* o valor 0.

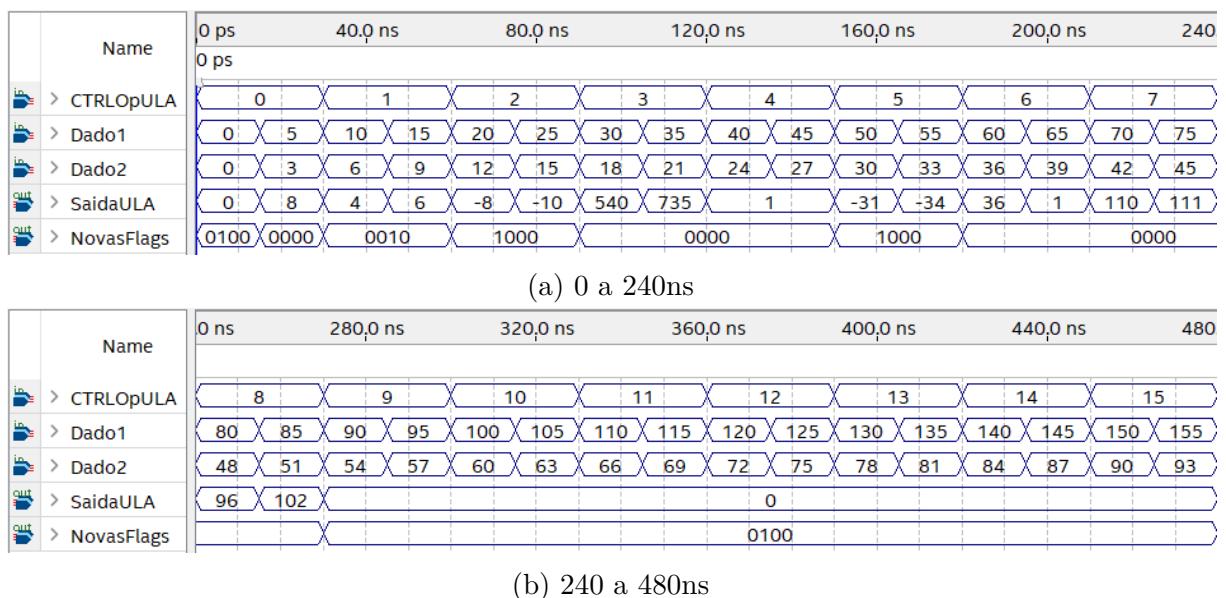


Figura 33 – Simulação de formas de onda da Unidade Lógica e Aritmética (ULA) gerada no software Quartus Prime.

5.6 Current Program Status Register (CPSR)

A Figura 34 mostra a simulação de formas de onda correspondente às entradas e saídas do CPSR, de acordo com o código da Figura 15. Nota-se que, entre 0 e 160ns, quando o sinal *CTRLEscritaCPSR* está ativo, a saída *FlagsNZCV* é atualizada a cada borda de subida do *clock*. Entre 160 e 320ns, quando o sinal está desativado, apesar da entrada *NovasFlags* mudar, a saída não é atualizada. Ela só volta a ser atualizada após 320ns, quando *CTRLEscritaCPSR* volta a estar ativo. Dessa maneira, constata-se o funcionamento adequado deste registrador.

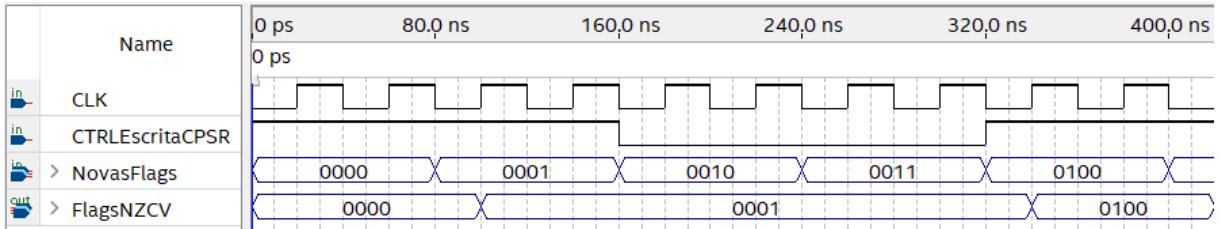


Figura 34 – Simulação de formas de onda do *Current Program Status Register* (CPSR) gerada no *software* Quartus Prime.

5.7 Memória de Dados

A Figura 35 mostra a simulação de formas de onda correspondente ao módulo da memória de dados desenvolvido em Verilog, conforme mostra a Figura 16.

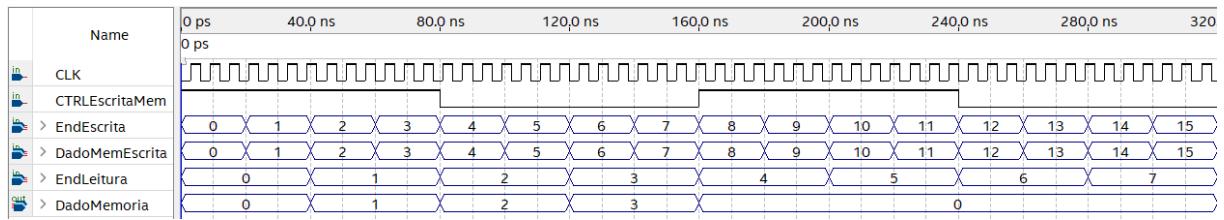


Figura 35 – Simulação de formas de onda da memória de dados gerada no *software* Quartus Prime.

Entre 0 e 80ns, enquanto o sinal de controle *CTRLEscritaMem* está ativo, dados são escritos nos endereços de memória 0 a 3. Isso é comprovado no intervalo de 0 a 160ns, quando a entrada *EndLeitura* leva à saída do módulo os dados previamente escritos nesses espaços da memória. Já quando o sinal *CTRLEscritaMem* está desativado, entre 80 e 160ns, os espaços de memória 4 a 7 não recebem novas informações. Analogamente, isso pode ser comprovado ao se observar o intervalo 160 a 320ns, onde o valor da saída é zero, o que não corresponde aos dados que teriam sido escritos nesses espaços da memória caso o sinal estivesse ativo. Assim, conclui-se que a memória de dados funciona de acordo com o previsto.

5.8 Multiplexador (MUX)

A Figura 36 mostra a simulação de formas de onda gerada a partir da implementação em Verilog do MUX, vista na Figura 17. É possível perceber que a cada 20ns a saída assume um valor entre os quatro de entrada, determinada pela entrada de seleção. Assim, conclui-se que o MUX implementado funciona de maneira adequada.

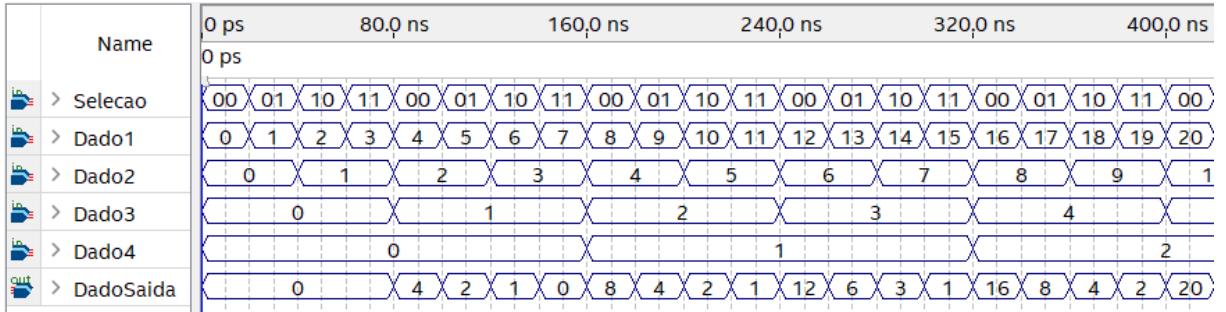


Figura 36 – Simulação de formas de onda do multiplexador (MUX) gerada no *software* Quartus Prime.

5.9 Adder

A Figura 37 mostra a simulação de formas de onda gerada a partir da implementação em Verilog do *adder*, visto na Figura 18. A função desse módulo é simplesmente somar as suas duas entradas *e*, portanto, é possível dizer que o módulo funciona como esperado.

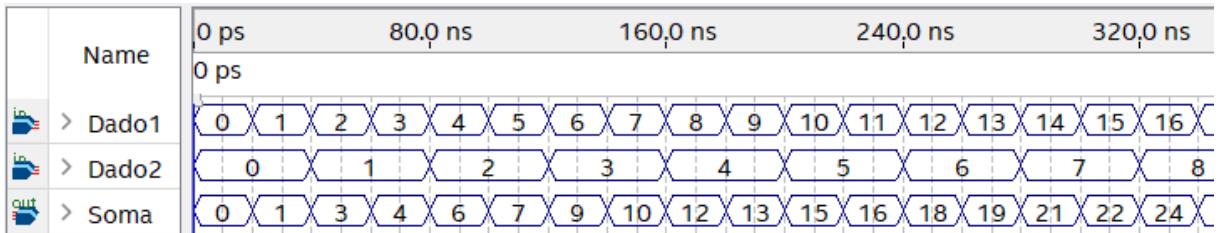


Figura 37 – Simulação de formas de onda do somador ou *adder* gerada no *software* Quartus Prime.

5.10 Unidade de Validação

A Figura 38 mostra a simulação de formas de onda da unidade de validação para todos os casos de campos *cond* e *flags*. Para cada caso, é possível identificar que a saída *Inibe* só é ativada quando as *flags* não estão de acordo com o pressuposto pelo campo *cond*, conforme visto na Figura 2.

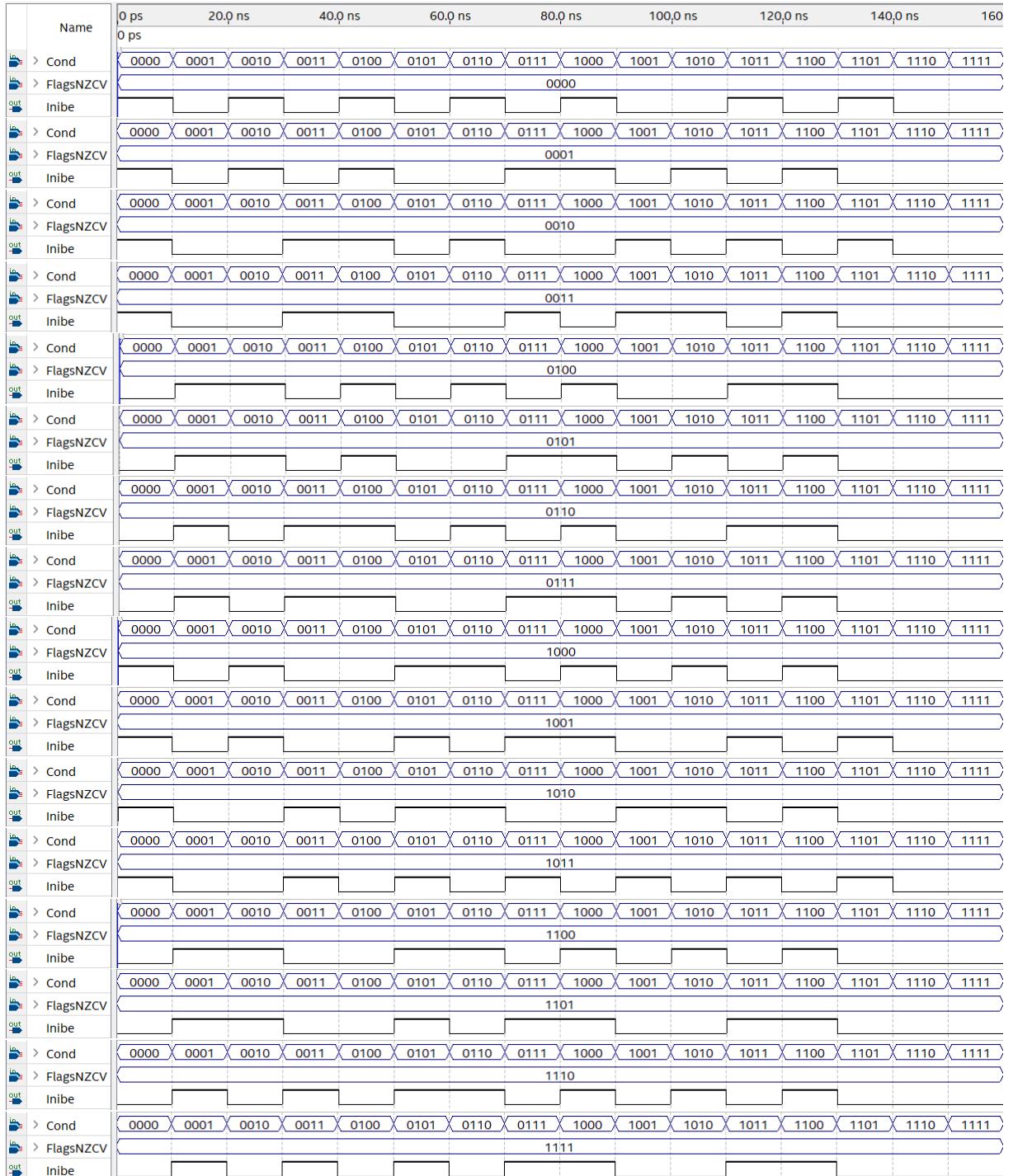


Figura 38 – Simulação de formas de onda da unidade de validação gerada no software Quartus Prime.

5.11 Integração Banco de Registradores e ULA

A simulação de formas de onda para a integração do banco de registradores com a ULA pode ser observada na Figura 39. Entre 0 a 320ns, quando o sinal *RegWrite* está ativo, dados são escritos em todos os registradores do banco, ou seja, em R0 a R31. Assim,

espera-se que o resultado *SaidaULA* esteja de acordo com esses valores em todo o intervalo demonstrado, ou seja, de 0 a 640ns.

Entre 0 e 270ns, é possível observar os resultados da ULA para as operações soma, subtração (das duas maneiras definidas), multiplicação, divisão, *not*, *and*, *or* e *xor*, codificadas pelo sinal *CTRLOpULA* de acordo com a Tabela 10. O mesmo acontece para alguns valores de *CTRLOpULA* no intervalo de 480 a 640ns. Em ambos os casos, os valores estão de acordo com os valores contidos nos registradores de entrada *RegLeitura1* e *RegLeitura2*. Além disso, entre 270 e 480ns, quando a operação da ULA cai em modo *default*, o resultado é 0. Assim, fica evidente o comportamento adequado da integração dos módulos.

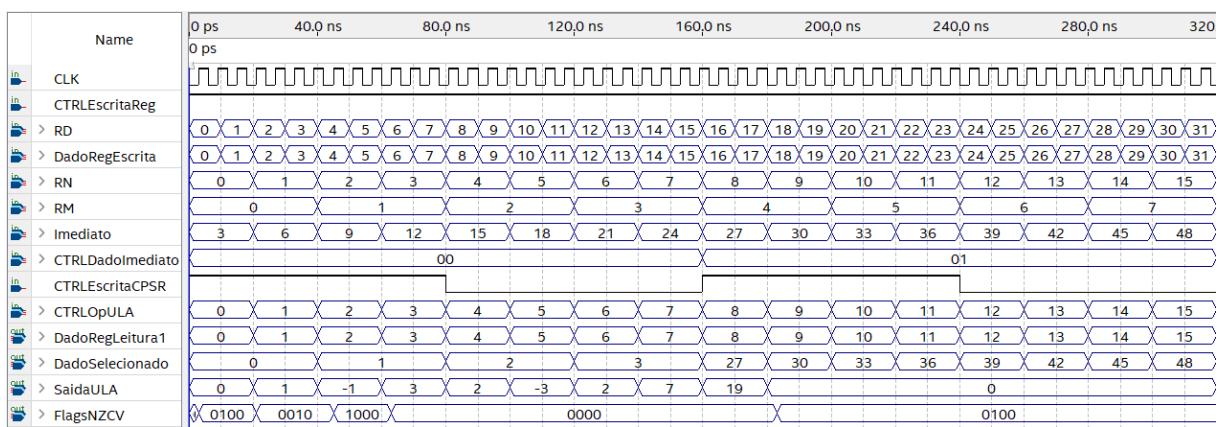


Figura 39 – Simulação de formas de onda do módulo de integração do banco de registradores, da Unidade Lógica e Aritmética (ULA) e do *Current Program Status Register* (CPSR) gerada no *software* Quartus Prime.

5.12 Entrada e Saída (E/S)

A Figura 40 mostra a simulação de formas de onda do módulo de entrada. É possível perceber que, toda vez que há uma mudança na entrada *Chaves* do módulo, a saída *Dado* é atualizada com o valor estendido para 32 bits da entrada. Assim, comprova-se o funcionamento esperado desse módulo.

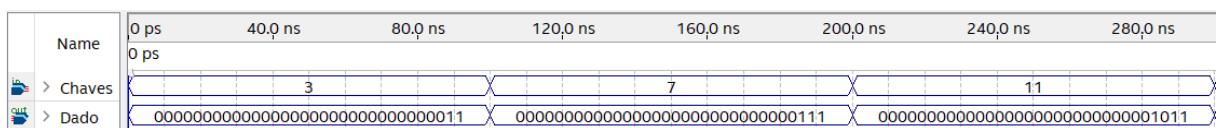


Figura 40 – Simulação de formas de onda do módulo de entrada.

Já o módulo de saída também funciona de maneira esperada. A sua simulação de formas de onda, que pode ser observada na Figura 41, mostra que a saída *Segmentos*, só é atualizada com a borda de subida do *clock*, quando o sinal de controle *CTRLSaida* está ativo. Percebe-se, também, que a string binária gerada corresponde ao código para display de 7 segmentos do valor 13.

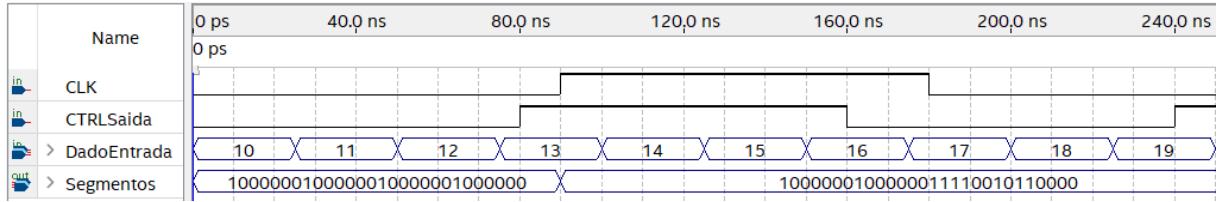
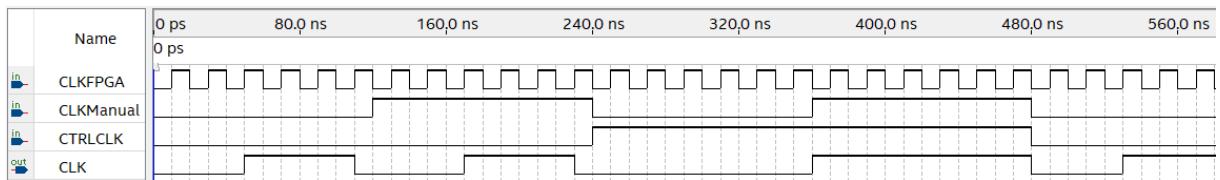


Figura 41 – Simulação de formas de onda do módulo de saída.

5.13 Seletor de Clock

A Figura 42 mostra a simulação de formas de onda da unidade de seleção de *clock*. Para fins da simulação, o contador implementado para a divisão de frequência foi alterado, e conta até o valor decimal 2. É possível notar que quando o sinal de controle *CTRLCLK* está desativado, a saída *CLK* assume o valor do *clock* dividido de *CLKFPGA*. Já quando está ativo, assume o valor de *CLKManual*. É interessante notar, também, que na transição de descida de *CTRLCLK* o valor do contador é zerado, o que evita que a próxima transição de subida de *CLK* ocorra imediatamente. Assim, é possível concluir que o módulo de seleção de *clock* funciona adequadamente.

Figura 42 – Simulação de formas de onda do módulo de seleção de *clock*.

5.14 Integração Completa

Para a verificação do funcionamento do processador e de sua interface com a placa FPGA, foram desenvolvidos dois códigos teste, que podem ser observados na Figura 43.

O primeiro código, da Figura 43a, calcula a média aritmética sobre uma quantidade de valores determinada pelo usuário. Esse código possui todos os tipos de instruções, assim como duas instruções sujeitas à execução condicional. Já o segundo código, da Figura 43b, calcula a área de um triângulo.

0	NOP		11101100
1	LDR	R0, #0	1110100100
2	LDR	R1, #0	1110100100000000001000
3	IN	R2	1110110010000100001000
4	NOP		11101100
5	OUT	R2	1110110011000100
6	BL	=14	1110011011111000
7	CMP	R2, #0	111000101100000100
8	UDIVNE	R0, R0, R2	00010000011000
9	MOV	R4, #3	1110001010110000001000
10	STR	R0, [R4, #1]	1110101110010001
11	LDR	R5, =4	1110100000000000101000
12	OUT	R5	111011001100101000
12	HLT		111011000100
14	CMP	R1, R2	11100000110000001000
15	BXGE	R31	1010010111111000
16	IN	R3	111011001000011000
17	NOP		11101100
18	OUT	R3	111011001100011000
19	ADD	R0, R0, R3	111000
20	ADD	R1, R1, #1	11100010000000001000010001
21	B	=14	11100100

(a) Código teste de cálculo de média aritmética.

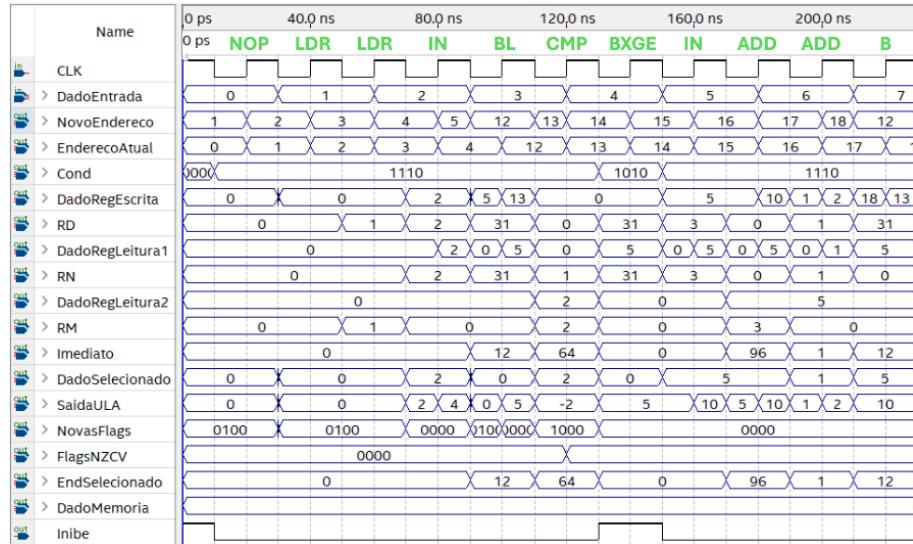
0	NOP		11101100
1	IN	R0	111011001000
2	NOP		11101100
3	OUT	R0	111011001100
4	IN	R1	111011001000001000
5	NOP		11101100
6	OUT	R1	111011001100001000
7	MUL	R2, R0, R1	111000000101000000001000
8	UDIV	R2, R2, #2	111000100110000100001000
9	OUT	R2	1110110011000100
10	HLT		111011000100

(b) Código teste de cálculo de área de triângulo.

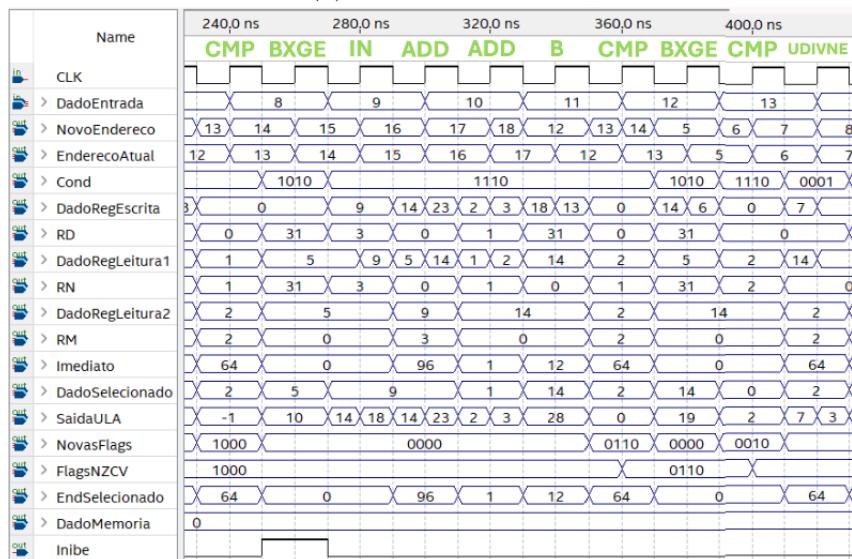
Figura 43 – Códigos teste. Fonte: A autora (2025).

5.14.1 Sem Interface com FPGA

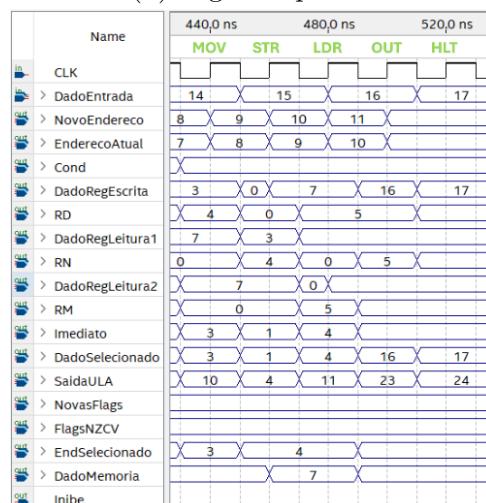
A Figura 44 mostra a simulação de formas de onda de uma versão antiga do código da Figura 43a. Nessa versão, ainda não haviam as instruções NOP e OUT após cada instrução IN.



(a) Primeira parte.



(b) Segunda parte.



(c) Terceira parte.

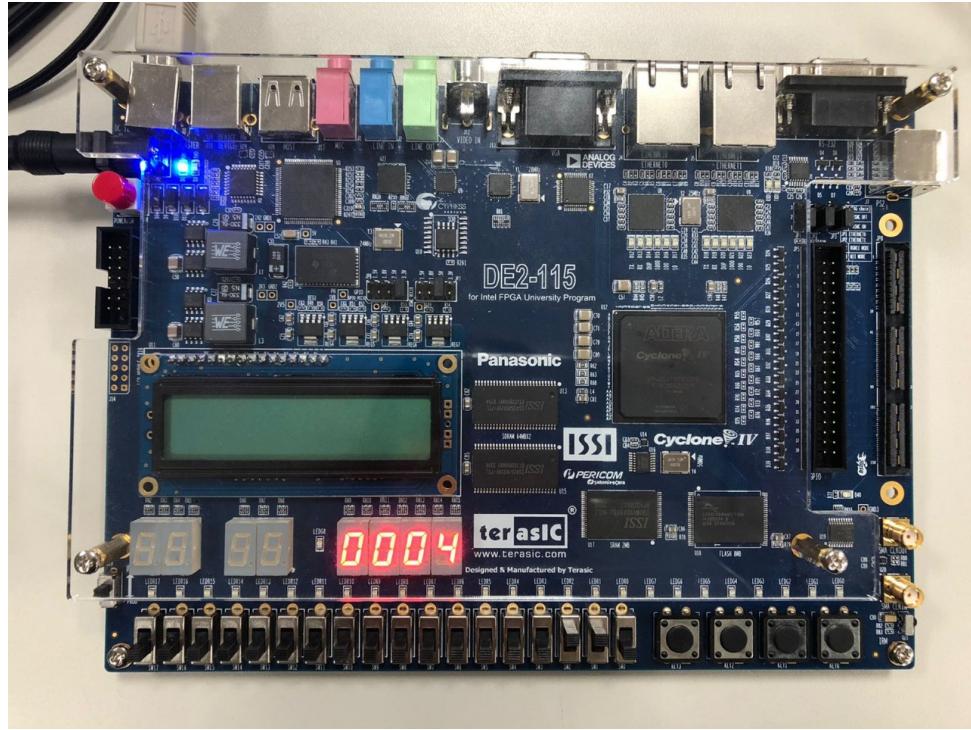
Figura 44 – Simulação de formas de onda do código teste de cálculo de média aritmética.

Na primeira parte do código, conforme mostra a Figura 44a, a instrução IN salva o valor de *DadoEntrada*, que é 2, no registrador R2. Assim, o processador repete a subrotina de soma, linhas 14 a 21, duas vezes, acumulando a soma dos valores de entrada, 5 e 7, no registrador R0. Então, ao reiniciar a subrotina pela terceira vez, o valor do contador em R1 e o valor de R2 são iguais, o que faz com que a instrução BXGE não seja mais inibida. Então, ao voltar da subrotina, essa soma é dividida pelo valor guardado em R2, resultando em 7.

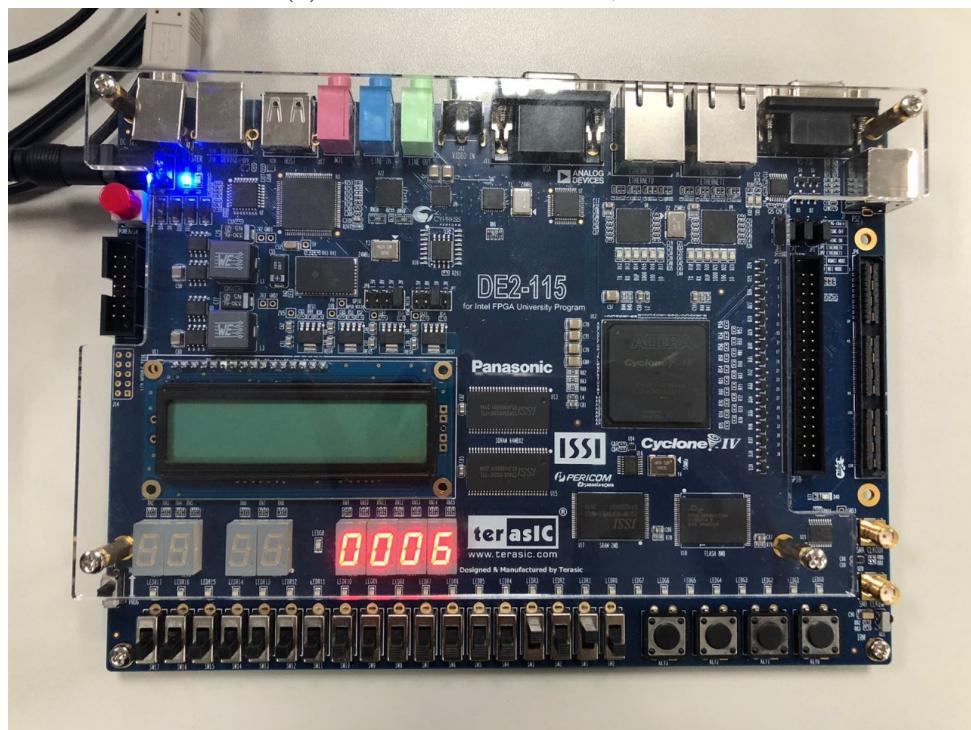
Além disso, o valor é guardado na memória por meio de uma instrução STR por deslocamento, e salvo novamente no banco de registradores, em R5, por uma instrução LDR direta. Ao final, uma instrução OUT faz o display do conteúdo de R5, que é 7. Portanto, evidencia-se o funcionamento da unidade integrada do processador ainda sem a interface com FPGA.

5.14.2 Com Interface com FPGA

As Figuras 45 e 46 mostram o display dos resultados de duas execuções dos códigos de cálculo de média aritmética e cálculo de área de triângulo, respectivamente.

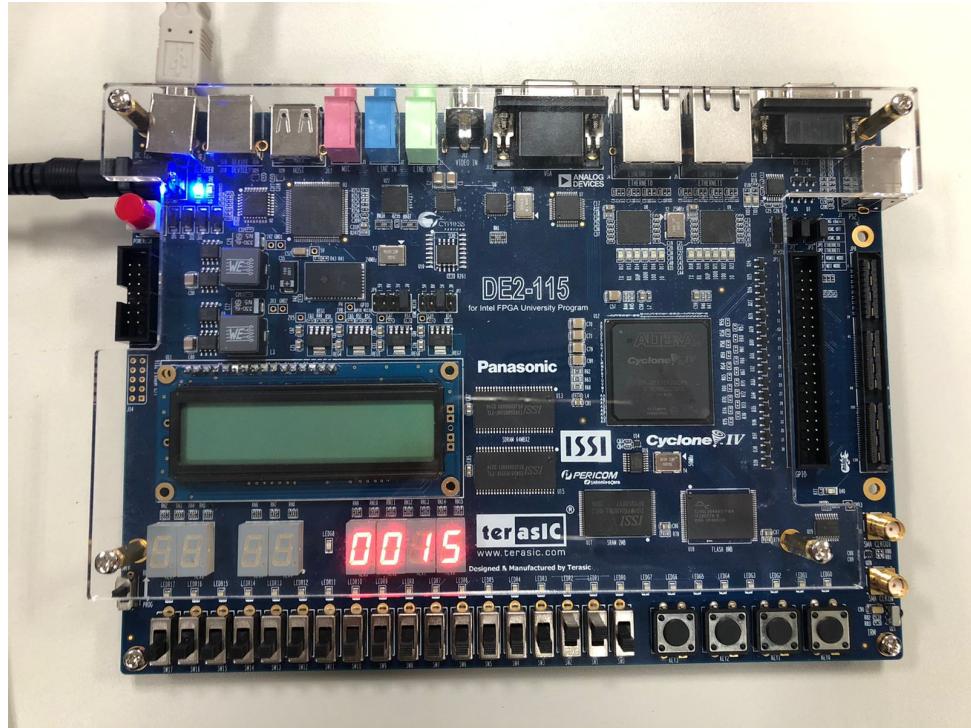


(a) Média aritmética de 0, 5 e 7.

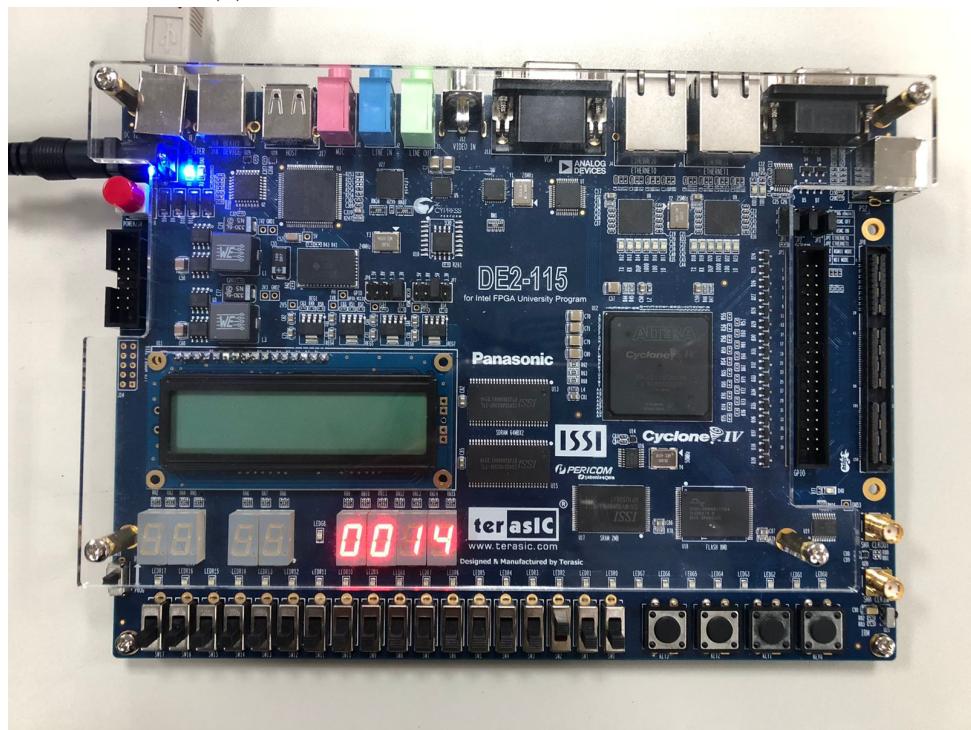


(b) Média aritmética de 4, 4, 6 e 10.

Figura 45 – Resultados do código de cálculo de média aritmética na placa FPGA.



(a) Área de triângulo com altura 5 e base 6.



(b) Área de triângulo com altura 7 e base 4.

Figura 46 – Resultados do código de cálculo de área de triângulo na placa FPGA.

Além disso, para o auxílio ao usuário, foi implementado um LED que acende quando uma instrução IN é identificada. A Figura 47 mostra isso em prática no FPGA.

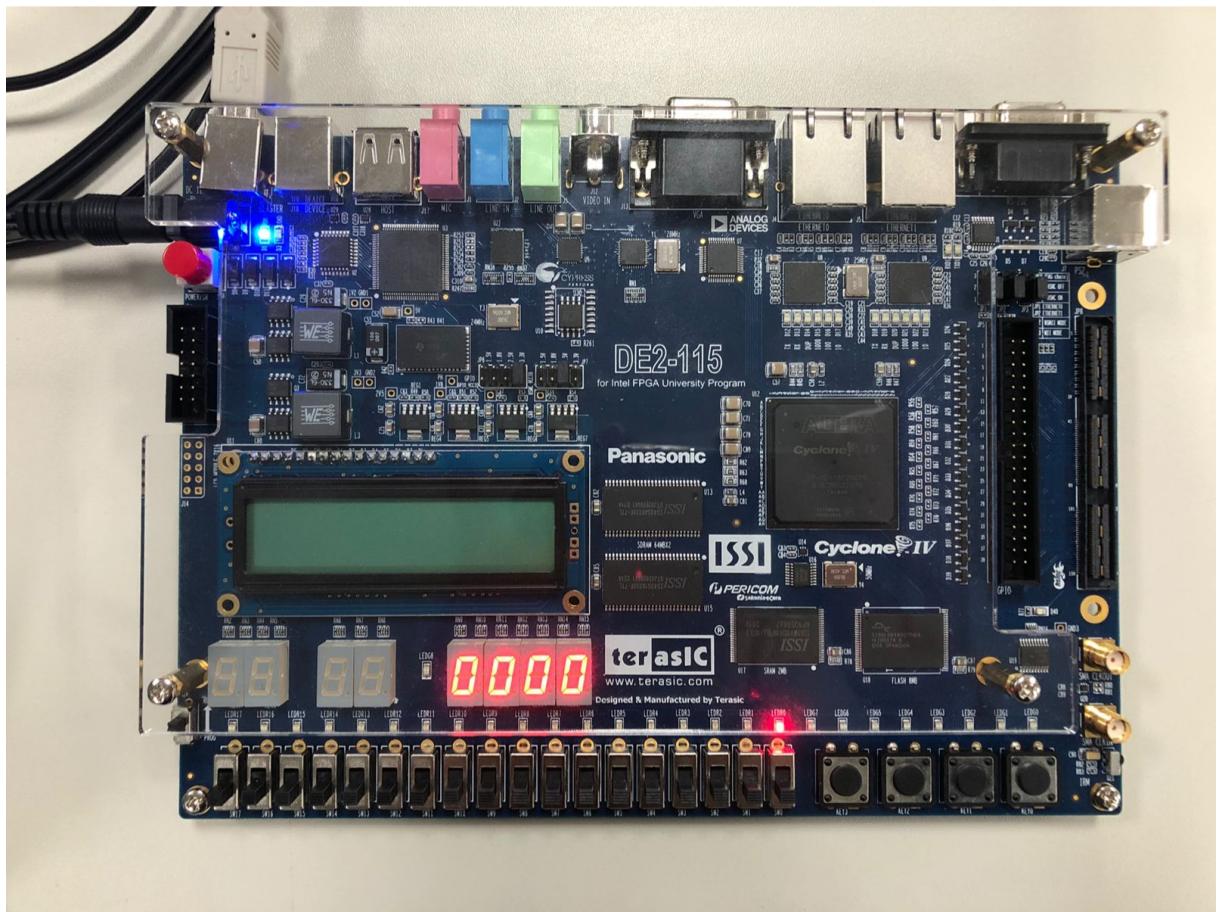


Figura 47 – LED durante execução de instrução IN.

6 Considerações Finais

Dessa maneira, foi apresentado o projeto e implementação em Verilog de um processador ARM de 32 bits, assim como de sua interface de comunicação com uma placa FPGA. Foram, inicialmente, discutidas as características arquiteturais do processador, como classificações da arquitetura, tamanho das instruções, quantidade de registradores e elementos estruturais gerais. Então, foi definido o conjunto de instruções adotado, seus tipos de instruções, formatos, modos de endereçamento à memória e, por fim, caminhos de dados. Conforme necessário, também foram descritas adaptações feitas na arquitetura ARM, como na quantidade de registradores de propósito geral, no CPSR ou no próprio conjunto de instruções.

Além disso, foram apresentadas as implementações em Verilog de todas as unidades funcionais do processador, assim como a da integração completa. No Capítulo 5, foram apresentadas as simulações de formas de onda de todas as implementações, de forma que evidenciou-se os seus funcionamentos adequados. Também foram apresentadas as simulações de formas de onda de códigos teste, assim como seus funcionamentos no FPGA.

Portanto, conclui-se que foi desenvolvido um processador funcional, capaz de realizar operações lógicas e aritméticas, de desvio, acesso à memória e comunicação externa. Assim, é possível dizer, também, que o projeto se manteve de acordo com os objetivos propostos no Capítulo 2.

Referências

ARM LIMITED. *ARM Architecture Reference Manual*. 5. ed. [S.l.], 2000. Disponível em: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/third-party/archives/ddi0100e_arm_arm.pdf>. Acesso em: 13 abril 2025. Citado 2 vezes nas páginas 14 e 19.

ARM LIMITED. *ARM Architecture Reference Manual: A-profile architecture*. 12. ed. [S.l.], 2024. Disponível em: <<https://developer.arm.com/documentation/ddi0487/latest/>>. Acesso em: 21 abril 2025. Citado 4 vezes nas páginas 3, 14, 15 e 19.

ARM LIMITED. *About Arm*. 2025. Disponível em: <<https://www.arm.com/company>>. Acesso em: 11 maio 2025. Citado na página 14.

IBM. *O que é FPGA (Field programmable gate array)?* 2024. Disponível em: <<https://www.ibm.com/br-pt/think/topics/field-programmable-gate-arrays>>. Acesso em: 5 junho 2025. Citado na página 17.

INSTITUTO FEDERAL DE SANTA CATARINA. *MCO018703 2020_2_AULA01: Arquiteturas Von-Neumann x Harvard*. 2020. Disponível em: <https://wiki.sj.ifsc.edu.br/index.php/MCO018703_2020_2_AULA01>. Acesso em: 14 maio 2025. Citado 2 vezes nas páginas 3 e 13.

INTEL. *FPGA Basics*. 2025. Acesso em: 5 de junho 2025. Disponível em: <<https://www.intel.com/content/www/us/en/support/programmable/support-resources/fpga-training/getting-started.html#:~:text=FPGA%20is%20an%20acronym%20for,after%20a%20product%20is%20deployed.>> Citado na página 17.

STALLINGS, W. *Arquitetura e organização de computadores*. 10. ed. São Paulo, SP: Pearson, 2017. E-book. Disponível em: <<https://plataforma.bvirtual.com.br>>. Acesso em: 10 maio 2025. Citado 5 vezes nas páginas 3, 12, 13, 15 e 16.

TOSTA, T. A. A. *Arquitetura e Organização de Computadores (Universidade Federal de São Paulo): AOC - Aula 2 - Conjuntos de Instruções*. 2024. Slides de aula. Disponível em: <<https://lipai.unifesp.br/teaching>>. Acesso em: 14 maio 2025. Citado na página 13.