

Ícaro Travain Darwich da Rocha

156307

Design e Implementação de um Processador ARM com a Utilização de um kit FPGA

São José dos Campos - Brasil

Setembro de 2024

Ícaro Travain Darwich da Rocha
156307

Design e Implementação de um Processador ARM com a Utilização de um kit FPGA

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

Docente: Prof. Dr. Sérgio Ronaldo Barros dos Santos

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Setembro de 2024

Resumo

O computador evoluiu por três etapas principais: válvulas, transistores e, finalmente, circuitos integrados, possibilitando a criação de organizações mais complexas com inúmeras portas lógicas. O objetivo do projeto atual é implementar um processador funcional com arquitetura ARM utilizando a linguagem de descrição de Hardware Verilog, para posterior implementação em um kit FPGA. O processador adota uma arquitetura RISC do tipo Harvard, com instruções de 32 bits e 5 bits reservados para registradores comuns. Foram implementados os registradores especiais CPSR e Link, além dos demais no banco de registradores. As instruções foram categorizadas em quatro tipos: processamento de dados, transferência de dados, desvio e uso diverso. A implementação foi realizada em Verilog de forma individual para cada módulo, que foram posteriormente integrados em um programa principal. As formas de onda analisadas mostraram que cada módulo e suas integrações funcionaram corretamente, o que foi confirmado ao unir todos os módulos e testar o programa no kit FPGA. O projeto atingiu seus objetivos iniciais, embora haja pontos a serem melhorados, como a otimização do código e a adição de novas funcionalidades ao CPSR.

Palavras-chaves: Arquitetura e Organização de Computadores. ARM. Kit FPGA. Verilog.

Listas de ilustrações

Figura 1 – Produção de chips	8
Figura 2 – Arquitetura de Von Neumann e de Harvard	12
Figura 3 – Tipos de Instrução ARM	13
Figura 4 – Sumário do campo COND	15
Figura 5 – Kit FPGA	17
Figura 6 – Datapath	23
Figura 7 – Datapath das instruções de processamento de dados	24
Figura 8 – Datapath das instruções de transferência de dados	25
Figura 9 – Datapath das instruções de desvio	26
Figura 10 – Datapath das instrução OUT	27
Figura 11 – Datapath das instrução IN	28
Figura 12 – ULA	29
Figura 13 – Banco de registradores	31
Figura 14 – Memória de dados (RAM).	32
Figura 15 – Memória de instruções (ROM).	33
Figura 16 – Unidade de instrução.	36
Figura 17 – Forma de onda para o PC.	55
Figura 18 – Forma de onda para operações aritméticas na ULA.	55
Figura 19 – Forma de onda para operações lógicas na ULA.	56
Figura 20 – Forma de onda multiplexador PC.	56
Figura 21 – Forma de onda memória RAM.	56
Figura 22 – Forma de onda do banco de registradores.	57
Figura 23 – Arquivo .txt de entrada	57
Figura 24 – Forma de onda memória ROM	57
Figura 25 – Forma de onda para a unidade de instrução.	58
Figura 26 – Forma de onda da Unidade de Controle.	58
Figura 27 – Forma de onda para o módulo de saída	59
Figura 28 – Forma de onda para o módulo de entrada	59
Figura 29 – Forma de onda para o módulo bcd	59
Figura 30 – Forma de onda para unidade de divisão do clock	60
Figura 31 – Forma de onda para o programa da área de um triângulo	61
Figura 32 – Forma de onda para o programa: Área de retângulos	62
Figura 33 – Tempo de compilação do programa: Área de retângulos	62

Lista de tabelas

Tabela 1 – Formato do CPSR	18
Tabela 2 – Formato das instruções de processamento de dados	19
Tabela 3 – Instruções de Processamento de Dados	19
Tabela 4 – Formato da instrução LDR	20
Tabela 5 – Formato da instrução STR	20
Tabela 6 – Instruções de Transferência de Dados	20
Tabela 7 – Formato das instruções B	21
Tabela 8 – Formato das instruções BI	21
Tabela 9 – Instruções de desvio	21
Tabela 10 – Formato das outras instruções	21
Tabela 11 – Outras Instruções	22
Tabela 12 – Operações da ULA e seus respectivos sinais de controle	29
Tabela 13 – Formato do sinal de controle	43
Tabela 14 – Operações do sinal SelClock	44

Sumário

1	INTRODUÇÃO	7
2	OBJETIVOS	9
2.1	Geral	9
2.2	Específico	9
3	FUNDAMENTAÇÃO TEÓRICA	10
3.1	Classificações de arquiteturas	10
3.1.1	Arquitetura CISC	10
3.1.2	Arquitetura RISC	10
3.1.3	Arquitetura de Von Neumann e Harvard	11
3.2	Processador ARM	12
3.2.1	Campo de identificação do tipo de instrução	12
3.2.2	Modos de Endereçamento	13
3.2.2.1	Endereçamento imediato	13
3.2.2.2	Endereçamento Direto	14
3.2.2.3	Endereçamento Direto Por deslocamento	14
3.2.3	Registradores Especiais	14
3.2.4	Campo COND	14
3.2.5	Arquitetura básica ARM	15
3.3	Linguagem de Descrição de Hardware	16
3.4	Field Programmable Gate Array (FPGA)	16
4	DESENVOLVIMENTO	18
4.1	Características do Processador	18
4.2	Registradores especiais	18
4.3	Conjunto de Instruções	19
4.3.1	Instruções de Processamento de dados	19
4.3.2	Instruções de Transferência de Dados	20
4.3.3	Instruções de Desvio	20
4.3.4	Outras Instruções	21
4.4	Caminho de Dados	22
4.4.1	<i>DataPath</i> - Instruções de processamento de dados	23
4.4.2	<i>DataPath</i> - Instruções de transferência de dados	24
4.4.3	<i>DataPath</i> - Instruções de desvio	25
4.4.4	<i>DataPath</i> - Outras Instruções	26

4.5	Implementação do Projeto	28
4.5.1	Unidade Lógica e Aritmética (ULA)	28
4.5.2	Banco de registradores	30
4.5.3	Memória de dados (RAM)	32
4.5.4	Memória de Instruções (ROM)	33
4.5.5	Módulo BCD	34
4.5.6	Program Counter (PC)	35
4.5.7	Unidade de instrução	35
4.5.8	Multiplexadores	40
4.5.9	Unidade de controle	43
4.5.10	Módulo de entrada	47
4.5.11	Módulo de saída	48
4.5.12	Módulo de divisão do clock	48
4.5.13	Integração Final	50
5	RESULTADOS E DISCUSSÃO	55
5.1	Forma de onda	55
5.1.1	PC	55
5.1.2	Unidade Lógica e Aritmética (ULA)	55
5.1.3	Multiplexadores	56
5.1.4	Memória RAM	56
5.1.5	Banco de registradores	57
5.1.6	Memória de instruções (ROM)	57
5.1.7	Unidade de Instrução	58
5.1.8	Unidade de controle	58
5.1.9	Módulo de saída	58
5.1.10	Módulo de entrada	59
5.1.11	Módulo BCD	59
5.1.12	Unidade de divisão do clock	60
5.1.13	Programas	60
5.1.13.1	Área do triângulo	60
5.1.13.2	Área de retângulos	61
6	CONSIDERAÇÕES FINAIS	63
	REFERÊNCIAS	64

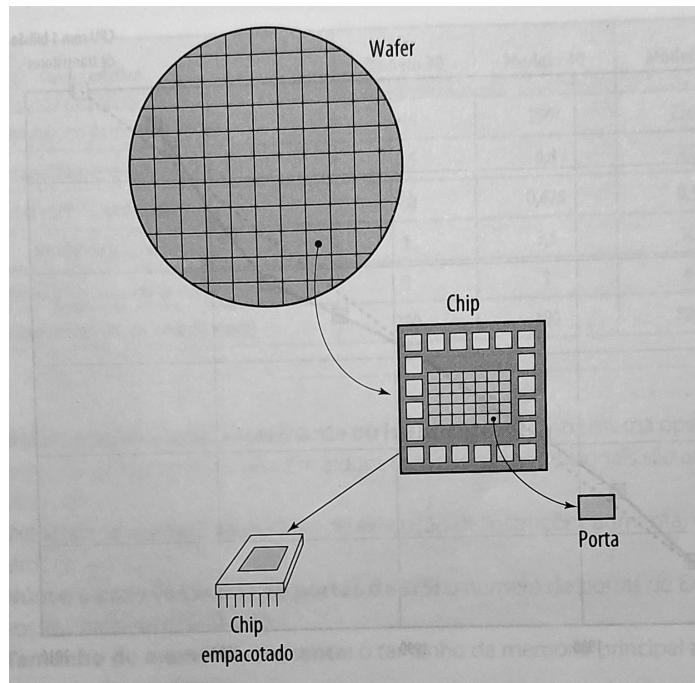
1 Introdução

O computador digital passou por três principais etapas para alcançar a sua versão mais atual. A etapa das válvulas teve início com o ENIAC (*Electronic Numerical and Computer*), um computador digital eletrônico projetado pelos EUA para suprir as necessidades durante a Segunda Guerra Mundial. No entanto, seu projeto só foi concluído em 1946, um ano após o término da guerra. Assim, ele foi utilizado em projetos posteriores, como a realização de cálculos para viabilizar uma bomba de hidrogênio.

A geração dos transistores veio para substituir as válvulas. Ao contrário das válvulas, que necessitavam de fios e cápsulas, os transistores são feitos de estado sólido, o que os torna muito mais compactos e eficientes na dissipação de calor. Essa mudança possibilitou um aumento de 5x na velocidade de processamento de 1957 até 1964. Além disso, foram introduzidas unidades lógicas e aritméticas e unidades de controle mais complexas.

Apesar da evolução, o processo de implementação dos transistores em máquinas mais complexas ainda era custoso, pois os componentes eram produzidos separadamente e depois soldados à placa. Para resolver esse problema, foram desenvolvidos os circuitos integrados. Os circuitos integrados aproveitam o fato de que componentes como resistor, transistor e condutor podem ser fabricados a partir do silício, possibilitando a criação de circuitos inteiros em um único pedaço de silício. O processo de produção de um chip pode ser visto na Figura 1, na qual um wafer de silício é dividido em pequenas áreas que são transformadas em chips, cada um com inúmeras portas lógicas. Por fim, o chip recebe um invólucro que protege as portas e oferece pinos de comunicação para a integração com circuitos maiores. ([STALLINGS, 2010](#))

Figura 1 – Produção de chips



Fonte: ([STALLINGS, 2010](#))

Essa evolução na capacidade de processamento possibilitou o surgimento de tecnologias que hoje fazem parte do cotidiano, como a internet, smartphones, dispositivos de casa inteligente, entre outros. Portanto, é fundamental que profissionais da área de Engenharia da Computação compreendam o funcionamento dos processadores desde sua base. Isso permite que a tecnologia seja aprimorada ao longo do tempo, possibilitando o desenvolvimento de novas tecnologias e melhorias contínuas.

2 Objetivos

2.1 Geral

Implementar um processador funcional em ARM que realize todas as operações definidas inicialmente, utilizando a bagagem teórica apresentada na disciplina de arquitetura e organização de computadores. O processador deverá ser sintetizado através do kit FPGA utilizando-se a linguagem de descrição de máquina Verilog.

2.2 Específico

Nesta seção, serão numerados cada um dos objetivos propostos para esse projeto:

1. Estudo e seleção do tipo de arquitetura que mais se encaixa com o projeto.
2. Selecionar as instruções a serem implementadas, assim como o seu tamanho e campos, visando o funcionamento correto do processador.
3. Estruturar um caminho de dados que implemente todas as funções definidas anteriormente.
4. Implementar as unidades do caminho do caminho de dados através da linguagem Verilog.
5. Implementar as unidades de entrada e saída no kit FPGA.
6. Testar as implementações para comprovar a correta implementação do processador.

3 Fundamentação Teórica

3.1 Classificações de arquiteturas

O primeiro ponto a ser abordado ao discutir processadores é sua arquitetura. Dois dos tipos mais relevantes de arquitetura são o RISC (Reduced Instruction Set Computer) e o CISC (Complex Instruction Set Computer). A seguir, serão exploradas suas características, vantagens e desvantagens.

3.1.1 Arquitetura CISC

A arquitetura CISC se destaca pela sua ampla variedade de instruções, capazes de executar várias tarefas em apenas um ciclo de *clock*. Instruções complexas, como manipulação de strings, operações com vetores e manipulação de pilhas, são características marcantes dessa arquitetura e raramente são encontradas em arquiteturas RISC. Essas instruções foram projetadas para suportar um alto nível de abstração, permitindo a realização de atividades complexas com um número reduzido de operações. ([STALLINGS, 2010](#))

Para acomodar essa diversidade de instruções, a arquitetura CISC geralmente utiliza uma variedade de modos de endereçamento, permitindo o acesso a diferentes tipos de dados. No entanto, apesar da flexibilidade proporcionada por esses modos de endereçamento, seu uso excessivo pode resultar em uma maior ineficiência energética quando comparado a arquiteturas mais simples, como a RISC.

3.1.2 Arquitetura RISC

A arquitetura RISC é conhecida por oferecer um conjunto reduzido de instruções, o que enfatiza sua simplicidade. Essa abordagem simplificada facilita tanto a decodificação quanto a organização do processador.

Ao contrário da arquitetura CISC, o pipeline é implementado de forma viável no RISC, permitindo a execução de instruções em paralelo. Isso otimiza o uso das unidades do processador e reduz o tempo de execução das instruções.

Devido à busca pela simplicidade nas instruções, torna-se necessário o uso de registradores especiais para armazenar resultados de operações. Um exemplo é o registrador HI no MIPS, que armazena o resto de uma divisão inteira.

A eficiência energética é outro aspecto relevante dessa arquitetura, especialmente devido à demanda por dispositivos portáteis. A arquitetura RISC se destaca por sua

capacidade de atender a essa necessidade.

Além disso, a adoção da arquitetura RISC tende a ser mais econômica em comparação com sua concorrente, devido ao menor custo de implementação do pipeline.

No entanto, apesar das vantagens, há alguns desafios inerentes a essa tecnologia. Primeiramente, há um maior consumo de memória RAM devido ao maior número de instruções necessárias para realizar uma determinada ação. Em segundo lugar, a escrita de instruções mais complexas pode ser mais desafiadora para o usuário, pois requer um maior número de instruções. Por fim, a implementação de compiladores pode ser mais complexa devido à natureza das instruções e ao seu conjunto reduzido.

3.1.3 Arquitetura de Von Neumann e Harvard

Outra maneira de classificar uma arquitetura é dividindo-a em duas categorias distintas: Von Neumann e Harvard, cada uma com características únicas.

A arquitetura de Von Neumann, conforme ilustrado na Figura 2, é composta por cinco componentes principais: a Unidade Lógica e Aritmética (ULA), a unidade de controle, a memória e os dispositivos de entrada e saída, além de seus respectivos barramentos. ((STALLINGS, 2010))

O primeiro conjunto é conhecido como CPU, que consiste na combinação da unidade de controle com a ULA. Sua função principal é coordenar as operações de processamento de dados, bem como realizar o próprio processamento.

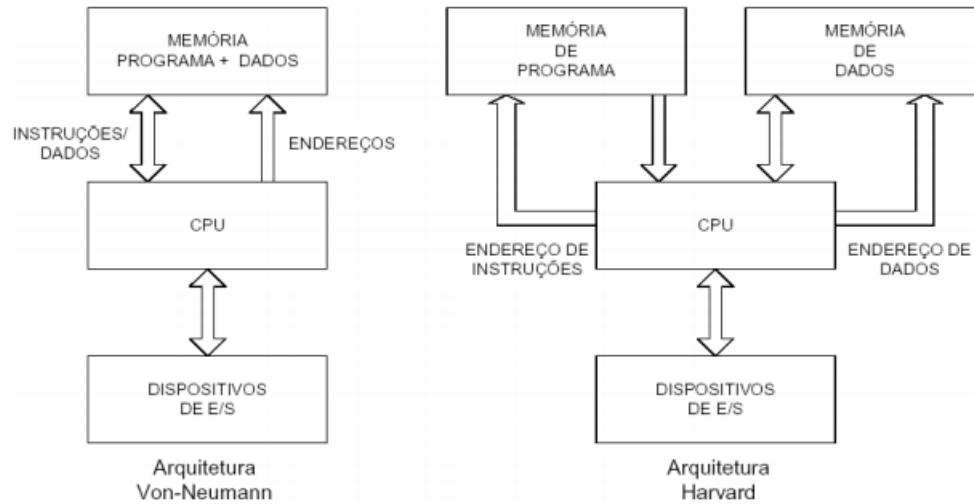
A memória é onde programas e instruções a serem executadas são armazenados. Os dispositivos de entrada e saída permitem a comunicação do computador com o mundo exterior, tanto recebendo dados quanto enviando.

Por último, os barramentos são necessários para transferir informações entre as unidades.

Entretanto, essa arquitetura apresenta um problema significativo: como a memória de instruções é compartilhada com a memória de dados, a unidade central acaba sobre carregada, afetando o desempenho do processamento de dados. Para superar essa limitação, foi desenvolvido o modelo de Harvard, mostrado na Figura 2. Nesse modelo, unidades separadas foram criadas para armazenar instruções e dados, cada uma com seus próprios barramentos.

Embora tenha resolvido o problema de gargalo, esse modelo resultou em custos adicionais devido à necessidade de duas memórias separadas.

Figura 2 – Arquitetura de Von Neumann e de Harvard

Fonte: ([IFSC, 2020](#))

3.2 Processador ARM

ARM, que significa Máquina RISC Avançada em inglês, é uma arquitetura de processadores baseada em RISC.

Os processadores ARM são empregados em uma variedade de dispositivos, incluindo smartphones, sistemas embarcados, dispositivos IoT (*Internet of Things*), computadores portáteis, entre outros. São valorizados pela sua combinação de desempenho e eficiência energética, o que os torna ideais para dispositivos móveis.

A arquitetura ARM é altamente flexível e, ao contrário de algumas concorrentes, não impõe regras rígidas para sua implementação. A ARM Holdings é a empresa detentora dos direitos de uso dessa arquitetura. Diferentemente de outras, ela não fabrica os chips, mas sim licencia a arquitetura para outras empresas, como Qualcomm, Apple, Samsung, entre outras.

Embora cada empresa tenha liberdade para projetar a implementação da tecnologia, algumas diretrizes de implementação podem ser encontradas no site da ARM Holdings, incluindo campos de identificação do tipo de instrução, campos de condição, registradores especiais e modos de endereçamento. ([ARM Limited, 2008](#))

3.2.1 Campo de identificação do tipo de instrução

Como os campos da instrução podem variar bastante dependendo da instrução a ser executada, é necessário que alguns bits da instrução sejam usados para esse propósito. Na figura 3, podem ser vistos os 3 primeiros bits após o campo 'cond', que determinam qual

formato da instrução será implementado. Note as similaridades nos formatos das instruções "Multiply" e "Multiply long", codificadas com 000. Ao mesmo tempo, as instruções "Data processing" codificadas por 001 possuem divergências, como a presença do Opcode.

Figura 3 – Tipos de Instrução ARM

Fonte: (ARM, 2000)

3.2.2 Modos de Endereçamento

Os modos de endereçamento são métodos utilizados para acessar a memória principal. Devido à limitação de bits em uma instrução, o endereço de memória de um dado específico frequentemente precisa ser referenciado em um espaço separado. Abaixo estão descritos os modos de endereçamento mais comuns:

3.2.2.1 Endereçamento imediato

Este é o tipo mais simples de endereçamento. Nele o endereço da memória fica armazenada diretamente na instrução. Ele é eficaz caso seja necessário acessar um endereço rapidamente, porém fica limitado a quantidade de bits do campo da instrução reservado para a tal finalidade.

3.2.2.2 Endereçamento Direto

Neste modo de endereçamento, se utiliza um registrador para armazenar o endereço da memória final. Neste caso, é ganhado uma versatilidade maior nos espaços de memória utilizados, uma vez que a quantidade de endereços possíveis para serem acessados não se restrigem a quantidade de bits no imediato, mas sim a quantidade de bits no registrador.

3.2.2.3 Endereçamento Direto Por deslocamento

Esse modo de Endereçamento combina o endereçamento direto e soma o endereço final com um deslocamento. Esta abordagem traz uma maior versatilidade no acesso dos dados.

3.2.3 Registradores Especiais

Registradores especiais têm funções específicas no processador. Eles podem ser implementados tanto em conjunto quanto separadamente dos demais e possuem instruções específicas para seu manuseio.

O CPSR (Current Program Status Register) é um registrador no processador ARM que contém informações sobre o estado atual do processador. Ele armazena flags de status, configurações do modo de operação e outros dados relevantes para o controle do fluxo do programa e a gestão das exceções. Algumas de suas flags são:

- **N(Negative)**: Indica se o resultado da operação registrada foi negativo.
- **Z (Zero)**: Indica se o resultado da última operação registrada foi zero.
- **C (Carry)**: Indica se a última operação registrada teve carry.
- **V (Overflow)**: Indica se o resultado da última operação registrada teve overflow.

O segundo registrador especial é chamado de *Link*. Ele é utilizado para armazenar a posição atual do PC (Program Counter) antes da execução de uma instrução de desvio. Esse armazenamento permite retornar à posição salva em um momento futuro.

3.2.4 Campo COND

O primeiro campo de uma instrução é denominado *cond* e define a condição para a execução da instrução. As condições atuais são armazenadas no CPSR (Current Program Status Register), conforme descrito anteriormente. A cada instrução executada, o CPSR é lido e comparado com a condição especificada na instrução. Se a condição for satisfeita, a instrução é executada; caso contrário, nada acontece. As possíveis condições estão apresentadas na figura 4.

Para exemplificar, se o sufixo de uma instrução terminar com *EQ*, isso indica que o campo *cond* dela é 0000. Nesse caso, a instrução será executada se a flag Z no CPSR for 1. No início do programa, as instruções devem ser utilizadas com o *cond always* até que a primeira instrução que modifica o CPSR seja empregada. ([Arm Limited, 2024](#))

Figura 4 – Sumário do campo COND

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Fonte: ([ARM, 2000](#))

3.2.5 Arquitetura básica ARM

Para que as instruções ARM sejam corretamente codificadas, são necessárias algumas estruturas básicas, como o Program Counter, o banco de registradores, a memória de instruções, a memória principal e a Unidade Lógica e Aritmética.

O Program Counter (PC) é um registrador que armazena a posição da próxima instrução a ser executada. Ele desempenha um papel fundamental no funcionamento dos computadores e, por vezes, é confundido com o próprio computador.

A Memória de Instruções armazena as instruções que serão executadas pelo processador. Essas instruções são recuperadas sequencialmente pelo Program Counter durante o ciclo de busca da instrução.

A Unidade Lógica e Aritmética (ULA) é responsável por realizar operações lógicas e aritméticas nos dados processados pelo processador. Ela executa operações como adição, subtração, operações lógicas (AND, OR, XOR), entre outras.

O Banco de Registradores é uma unidade que contém diversos registradores mapeados. Neles, são armazenados os resultados temporários das operações do processador, bem como endereços de memória. O tamanho do banco de registradores afeta diretamente os campos de instrução, uma vez que o acesso a ele é realizado de forma direta. Dessa forma, a quantidade máxima de registradores possíveis é 2^n , com n sendo a quantidade de bits do campo reservado aos registradores.

A memória Principal é onde são armazenados os dados que serão acessadas e processadas pelo processador. Diferente do banco de registradores, ele possui uma quantidade muito maior de endereços, sendo necessário outras abordagens para acessar toda a sua extensão.

A memória principal desempenha um papel crucial na computação, armazenando os dados que serão acessados e processados pelo processador. Ao contrário do banco de registradores, ela oferece uma capacidade significativamente maior de armazenamento, embora isso exija abordagens mais complexas para acessar toda a sua extensão.

3.3 Linguagem de Descrição de Hardware

Em eletrônica, HDL (*Hardware Description Language*) é uma linguagem de computador que ajuda a esclarecer a estrutura e desempenho de circuitos eletrônicos, principalmente circuitos digitais. O HDL permite uma explicação formal e específica do circuito eletrônico, o que possibilita a investigação automática na simulação de circuitos.

Diferentemente das linguagens de programação, uma linguagem de descrição de hardware serve para descrever circuitos fisicamente construídos. Entre os HDLs mais conhecidos estão o Verilog e o VHDL. Essas linguagens podem ser compiladas com o auxílio de um programa especializado, como o Quartus Prime, e o binário resultante pode ser utilizado em um kit, como o FPGA. Dessa forma, torna-se mais simples o teste de circuitos mais complexos, como um processador.

Além disso, essas linguagens podem ser utilizadas em conjunto com ferramentas de síntese de hardware, o que permite gerar o layout físico de um circuito.

3.4 Field Programmable Gate Array (FPGA)

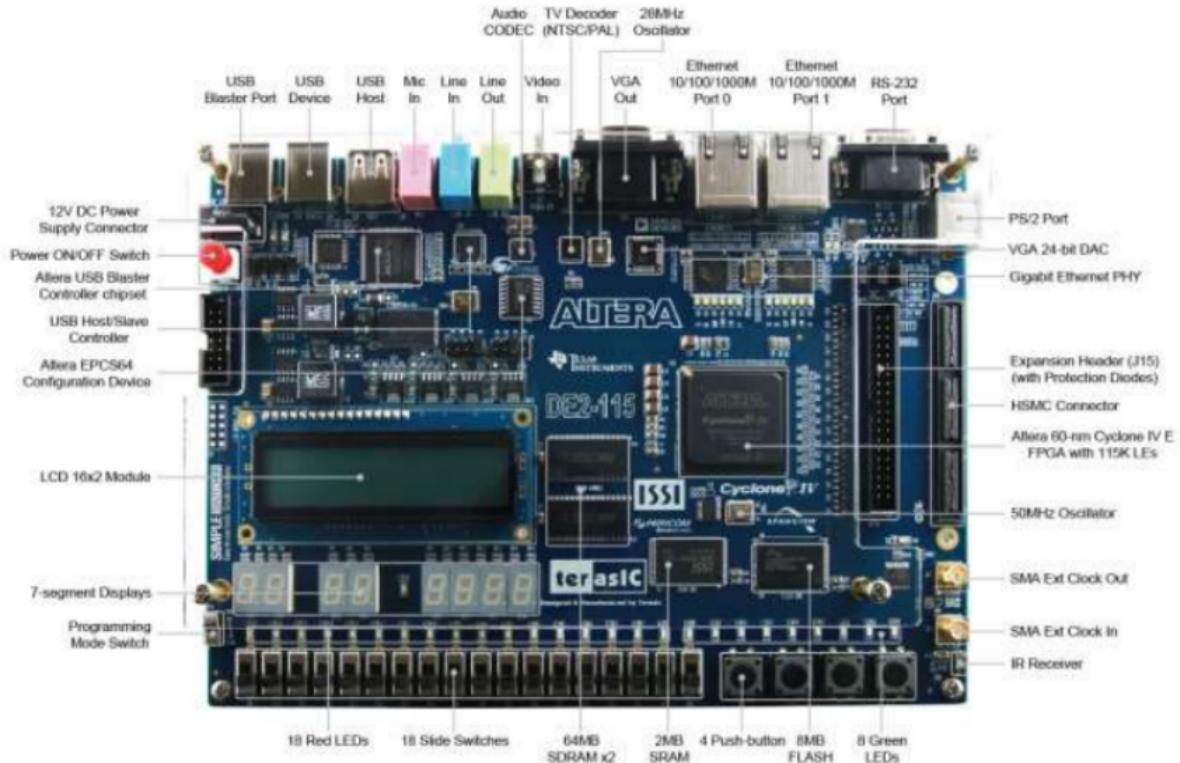
O *Field Programmable Gate Array* (FPGA) é um circuito integrado que se molda de acordo com a necessidade do usuário, ao contrário dos chips ASICs que vêm pré-programados, como os utilizados em microondas e geladeiras. Para realizar sua programação, é necessário utilizar uma HDL, como Verilog ou VHDL.

Devido à sua flexibilidade, o FPGA é perfeito para a implementação de projetos

que requerem um alto grau de especialização, como a implementação de um processador.

Ao longo de todo o projeto, será utilizado o kit FPGA da família *Cyclone IV*, modelo EP4CE115F29C7, mostrado na figura 5.

Figura 5 – Kit FPGA



Fonte: ([Altera Corporation, 2013](#))

4 Desenvolvimento

A partir dos conceitos desenvolvidos durante o embasamento teórico, será possível definir com precisão como será implementado o processador em ARM em linguagem de descrição de Hardware.

4.1 Características do Processador

1. O processador irá trabalhar com 32 bits. Deste modo, o banco de memória, banco de registradores e as instruções seguirão este padrão
2. O banco de Registradores terá 32 registradores. Assim, os campos de referenciamento dos registradores terá 5 bits.
3. Será realizado um modelo baseado em ARM, neste modo os padrões seguirão boa parte da lógica apresentada anteriormente.
4. Por ser um processsador ARM, a arquitetura seguirá o tipo RISC.
5. A arquitetura utilizada será do tipo Harvard.

4.2 Registradores especiais

Os registradores especiais *Link* e *CPSR* foram implementados junto aos outros registradores, correspondendo a R31 e R30, respectivamente.

Apesar do registradores possuírem 32 bits, apenas 4 bits do CPSR serão utilizados, sendo eles descritos na tabela 1.

Tabela 1 – Formato do CPSR

[3]	[2]	[1]	[0]
V	C	Z	N

Fonte: O Autor (2024)

O registrador *Link* utilizará os 32 bits para poder alcançar o maior número de posições possíveis.

Nesta implementação, os registradores especiais podem ser alterados tanto pelas instruções especiais quanto diretamente. Isso possibilita uma maior versatilidade na elaboração dos programas, pois caso estas fuções especiais não sejam utilizadas, os registradores ficam livres para outras operações.

4.3 Conjunto de Instruções

Na implementação do processador ARM serão utilizados quatro tipos de instruções: Instruções de processamento de dados, instruções de transferência de dados, instruções de desvio e outras instruções. Deste modo, são necessários 2 bits de distinção.

4.3.1 Instruções de Processamento de dados

O primeiro tipo de instrução corresponde ao processamento de dados e é identificado pela decodificação dos bits 00. O formato dessas instruções pode ser visto na Tabela 2.

Essa categoria de instrução é caracterizada pela presença de um Opcode que define o tipo de operação, um bit "I" para indicar a presença de um valor imediato, um bit "S" para determinar se o CPSR será alterado, dois campos para registradores e, por fim, o último campo pode ser um registrador ou um valor imediato, dependendo do bit "I". As possíveis instruções estão descritas na tabela 3, as quais dependem do OPcode, do campo I e do S. Qualquer instrução de processamento de dados pode conter um imediato.

Tabela 2 – Formato das instruções de processamento de dados

	[31:28]	[27:26]	[25]	[24:21]	[20]	[19:15]	[14:10]	[9:0]
Cond	Tipo Instrução	I	Opcode	S	Rn	Rd	Imm	

Fonte: O Autor (2024)

Tabela 3 – Instruções de Processamento de Dados

Opcode	I	S	Nome	Função
0000	0	0	AND	$rd \leftarrow rn \& rt$
0000	1	0	ANDI	$rd \leftarrow rn \& Imm$
0001	0	0	EOR	$rd \leftarrow rn EOR rm$
0010	0	0	SUB	$rd \leftarrow rn - rm$
0010	0	1	SUBS	$rd \leftarrow rn - rm$
0010	1	0	SUBI	$rd \leftarrow rn - Imm$
0011	0	0	ADD	$rd \leftarrow rn + rm$
0011	1	0	ADDI	$rd \leftarrow rn + Imm$
0100	0	0	MRS	$rn \leftarrow CPSR$
0101	0	1	MSR	$CPSR \leftarrow rn$
0110	0	1	TST	$CPSR \leftarrow rn AND rm$
0111	0	1	CMP	$CPSR \leftarrow rn - rm$
1000	0	0	ORR	$rd \leftarrow rn OR Rm$
1000	1	0	ORRI	$rd \leftarrow rn OR Imm$
1001	0	0	MOV	$rd \leftarrow Rm$
1001	1	0	MOVI	$rd \leftarrow Imm$
1010	0	0	MUL	$rd \leftarrow rn * rm$
1011	0	0	UDIV	$rd \leftarrow rn / rm$

Fonte: O Autor (2024)

4.3.2 Instruções de Transferência de Dados

As instruções de transferência de dados são empregadas para carregar ou copiar dados da memória. Elas são identificadas pela codificação dos bits 01 no campo "Tipo de instrução" e apresentam três bits de codificação.

O bit I indica se a instrução contém um valor imediato. O bit U seleciona a direção do deslocamento no modo de endereçamento direto por deslocamento, optando por posições menores ou maiores na memória. Por fim, quando o bit L/S é igual a 1, indica que a instrução é de carregamento (*LDR*), caso contrário, quando o bit L é igual a 0, indica uma instrução de armazenamento (*STR*).

O formato das instruções de load e store pode ser visto nas Tabelas 4 e 5, respectivamente. Nas instruções LDR, o campo [23] é 1, indicando uma instrução LDR. Nesse caso, o campo [17:13] apresenta o registrador no qual os dados serão carregados. Em contraste, nas instruções STR, o mesmo campo [17:13] representa o registrador de onde os dados serão retirados. Essas operações estão melhor representadas na Tabela 6, na qual a base representa o endereço da memória RAM.

Tabela 4 – Formato da instrução LDR

[31:28]	[27:26]	[25]	[24]	[23]	[22:18]	[17:13]	[12:0]
Cond	Tipo Instrução	I	U	1	Rn	Rd	Imm

Fonte: O Autor (2024)

Tabela 5 – Formato da instrução STR

[31:28]	[27:26]	[25]	[24]	[23]	[22:18]	[17:13]	[12:0]
Cond	Tipo Instrução	I	U	0	Rn	Rm	Imm

Fonte: O Autor (2024)

Tabela 6 – Instruções de Transferência de Dados

I	U	L	Nome	Função
1	0	1	LDR	$Rd \leftarrow Base[Rn + Imm]$
1	0	0	STR	$Base[Rn + Imm] \leftarrow Rm$

Fonte: O Autor (2024)

4.3.3 Instruções de Desvio

As instruções de desvio, codificadas como 10, incluem o campo I para indicar a presença de um valor imediato, junto com o bit L, que sinaliza a necessidade de atualizar o registrador "Link". Se o bit I no campo [25] for 1, o processador utilizará o formato da Tabela 7. Caso contrário, será usado o formato da Tabela 8.

Essas instruções alteram o valor do contador de programa (PC), o que resulta em uma mudança na posição atual da memória de instruções. As instruções implementadas estão mostradas na Tabela 9, na qual também pode ser observada a alteração no registrador Link.

Tabela 7 – Formato das instruções B

[31:28]	[27:26]	[25]	[24]	[23:0]
Cond	Tipo Instrução	1	L	Imm

Fonte: O Autor (2024)

Tabela 8 – Formato das instruções BI

[31:28]	[27:26]	[25]	[24]	[23:19]	[18:0]
Cond	Tipo Instrução	0	L	Rn	xxx

Fonte: O Autor (2024)

Tabela 9 – Instruções de desvio

I	L	Nome	Função
1	0	BI	PC ← Imm
1	1	BIL	PC ← Imm and L ← PC
0	0	B	PC ← Rn
0	1	BL	PC ← Rn and L ← PC

Fonte: O Autor (2024)

4.3.4 Outras Instruções

Outras instruções de uso geral são codificadas como 11. Elas possuem apenas um campo de OPcode, conforme ilustrado na tabela 10.

Dentre as instruções de uso geral, está a instrução NOP, que tem a finalidade de pular um ciclo de clock sem alterar o processador. A instrução IN indica que um dado será inserido no módulo de entrada. A instrução OUT envia o dado de um registrador específico para o módulo de saída. Por fim, a instrução FINISH encerra o programa. Todas essas instruções e seus respectivos opcodes podem ser vistos na tabela 11.

Tabela 10 – Formato das outras instruções

[31:28]	[27:26]	[25:23]	[22:0]
Cond	Tipo Instrução	Opcode	xxx

Fonte: O Autor (2024)

Tabela 11 – Outras Instruções

Opcode	Nome
000	NOP
001	IN
010	OUT
011	FINISH

Fonte: O Autor (2024)

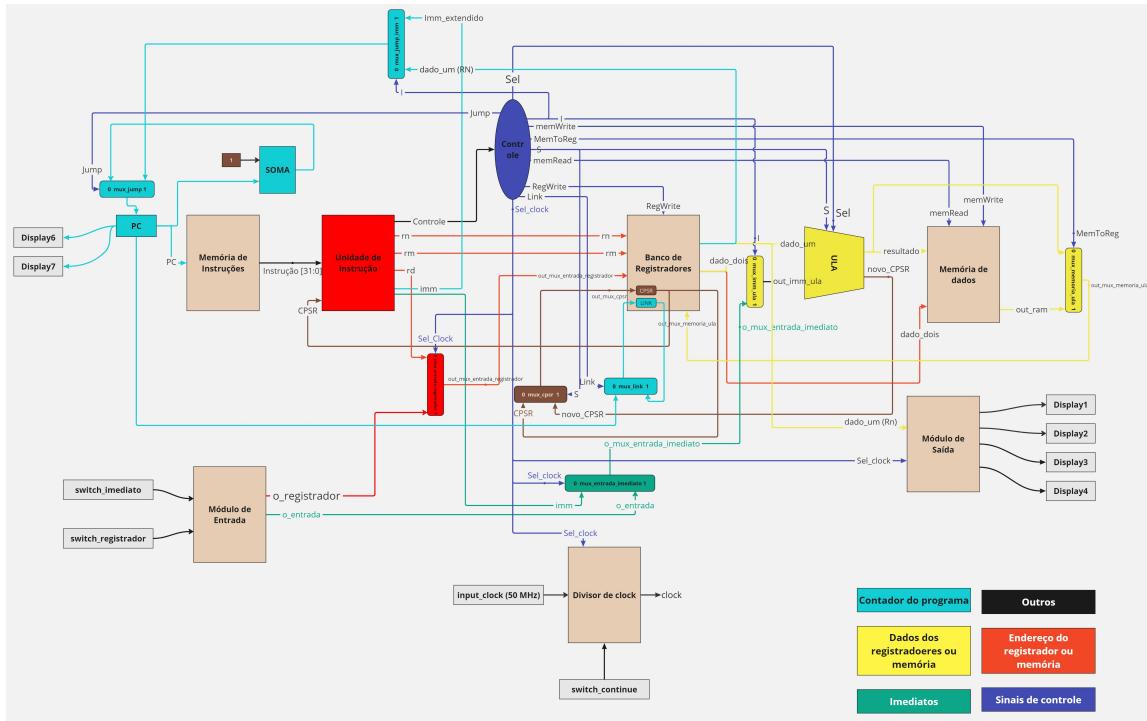
4.4 Caminho de Dados

Para garantir que as instruções sejam interpretadas corretamente pelo processador, é fundamental dispor de estruturas capazes de decifrá-las de maneira adequada. Para essa implementação, foi empregado o datapath representado na figura 6. Nesse esquema, destacam-se o Contador de Programa (PC), a memória de instruções, a Unidade de Instrução, a Unidade de Controle, o Banco de Registradores, a Unidade Lógica e Aritmética (ULA), a memória de dados, diversos multiplexadores que direcionam os dados pelo processador, o módulo de entrada, o módulo de saída e o módulo divisor do clock.

A unidade de instrução desempenha um papel intermediário crucial ao referenciar adequadamente os bits dos registradores para o banco de registradores. A estrutura é necessária devido aos campos das instruções, os quais não possuem uma estrutura fixa e podem variar conforme o tipo da instrução.

As cores na figura 6 representam os tipos de informação em cada trecho do caminho de dados, os seus significados estão descritos no canto inferior direito da imagem. Já os retângulos cinza são as saídas ou entradas que serão conectadas ao FPGA.

Figura 6 – Datapath



Fonte: O Autor (2024)

4.4.1 DataPath - Instruções de processamento de dados

Para que uma instrução do tipo de processamento de dados seja realizada tem-se os seguintes passos identificados na figura 7.

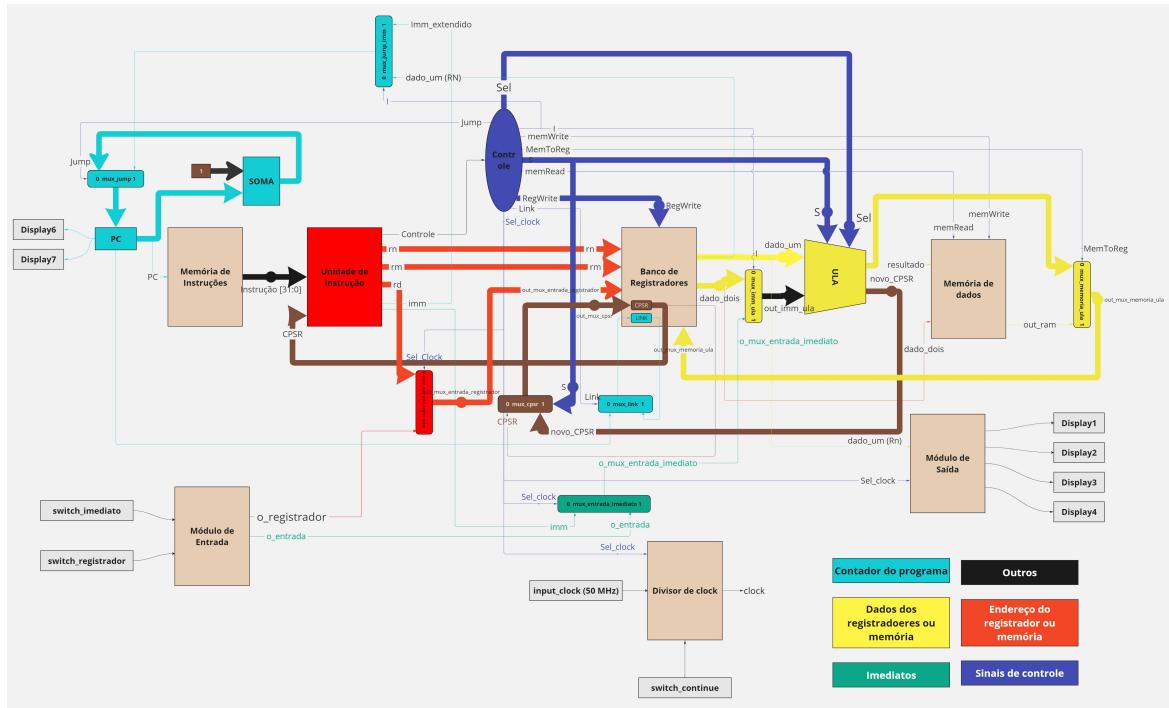
Do lado esquerdo observa-se um circuito sequencial, o qual sempre soma 1 ao PC para ser realizada a próxima instrução.

Do lado direito é visto uma instrução SUBSEQ sendo executada, ela sai da memória de instruções passa para a unidade de instrução na qual os bits que representam os bits dos endereços dos registradores são devidamente direcionados para os destinos. Os dados saem do banco de registradores são processados na ULA e depois escritos em algum registrador do banco, pois o *RegWrite* está ativo.

Como o campo cond está setado como EQ, a unidade de instrução só vai permitir que a instrução seja executada caso os valores do CPSR estejam com a mesma codificação.

Por fim, a unidade de controle manda a informação S para a ULA reescrever o dado presente no CPSR.

Figura 7 – Datapath das instruções de processamento de dados



Fonte: O Autor (2024)

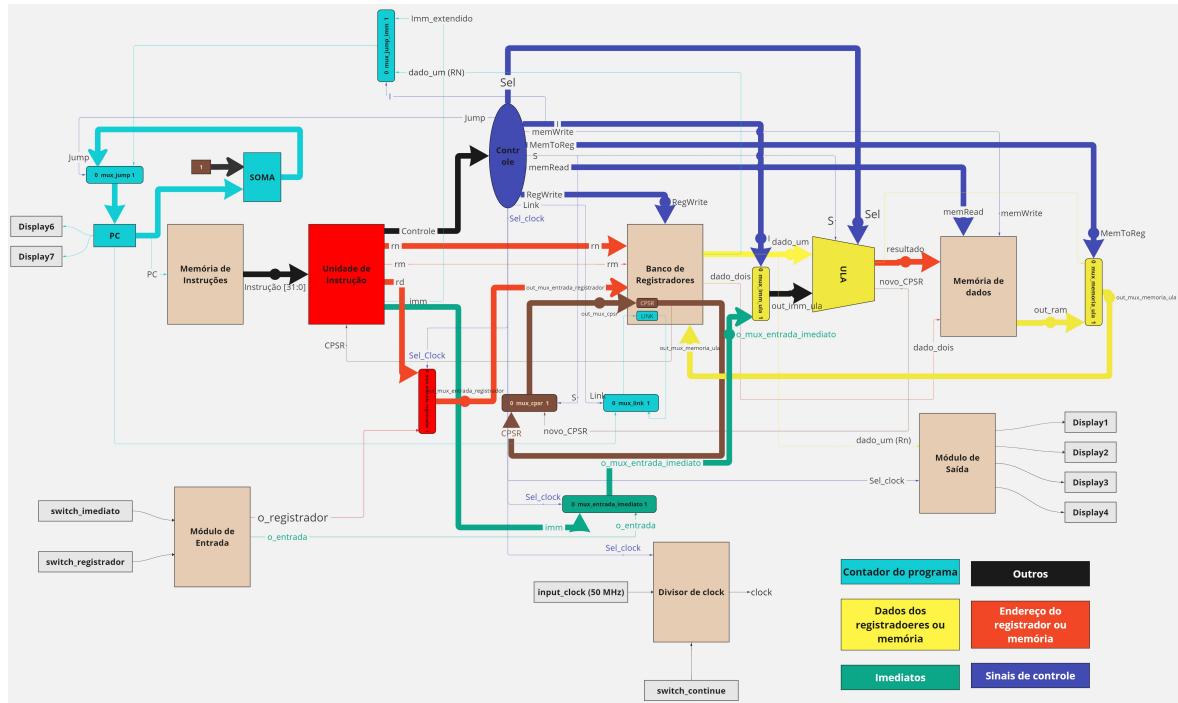
4.4.2 DataPath - Instruções de transferência de dados

A instrução LDREQ inicia-se com a memória de instrução enviando a instrução para a unidade de instrução que por sua vez distribui os sinais pelo processador caso esteja em conformidade com o CPSR. Rn e Ed são enviados para o banco de registradores, porém o valor do imediato é enviado diretamente para a ULA.

A soma entre o imediato e Rn é necessária para calcular o endereço da memória RAM e assim é enviada para a porta de endereço da memória principal. Com a ativação do sinal MemRead, torna-se possível a leitura do dado, que é então transferido para o registrador Rd.

A unidade de controle emite os comandos MemToReg para que a instrução lida na memória seja encaminhada para o registrador de destino, o sinal RegWrite, para permitir a escrita no registrador de destino, o sinal sel indica a operação de soma na ULA e o bit I indica a presença do imediato na instrução. Todo esse caminho de dados pode ser verificado na Figura 8.

Figura 8 – Datapath das instruções de transferência de dados



Fonte: O Autor (2024)

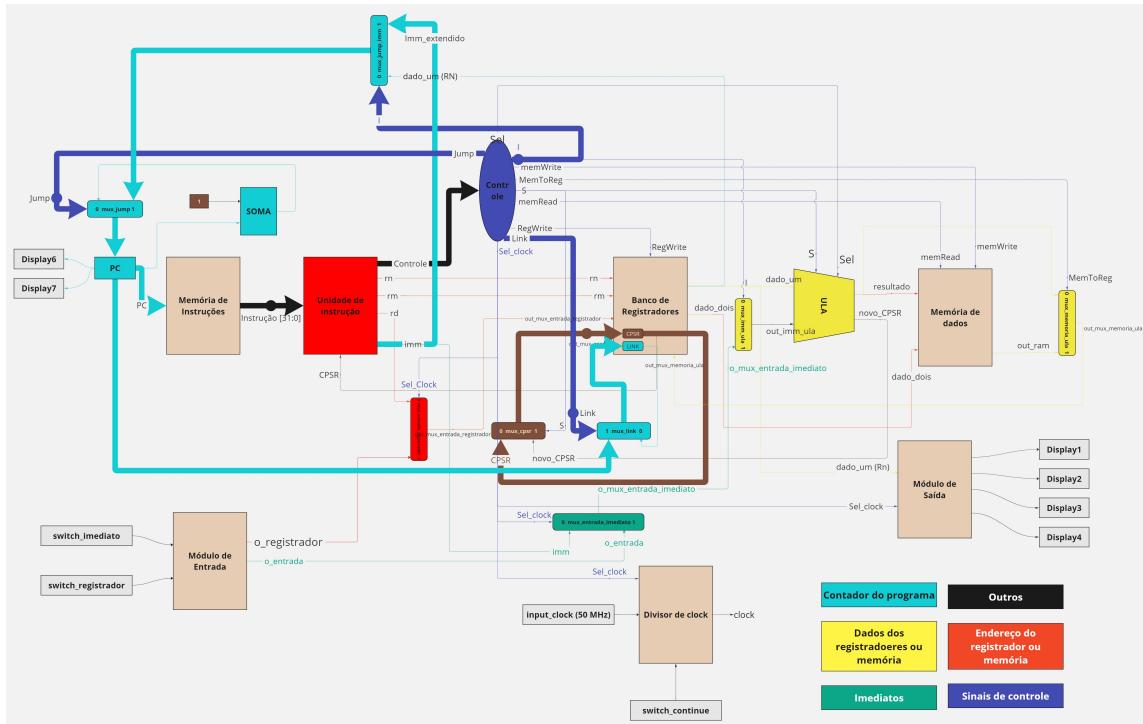
4.4.3 DataPath - Instruções de desvio

Diferente das anteriores, uma instrução de desvio impede a soma habitual de 1 ao PC.

No exemplo da figura 9 está sendo ilustrada a execução de uma instrução BIL, nesta instrução estão setadas as flags I e L, elas representam que um imediato está sendo utilizado e que o registrador especial link vai ser carregado com o valor atual do PC.

A unidade de controle manda os comandos de Jump e I. O comando Jump altera o multiplexador próximo ao PC, impedindo que ele seja incrementado de um. O I sinaliza para que o valor do imediato saia direto da unidade de instrução e vá para o PC.

Figura 9 – Datapath das instruções de desvio



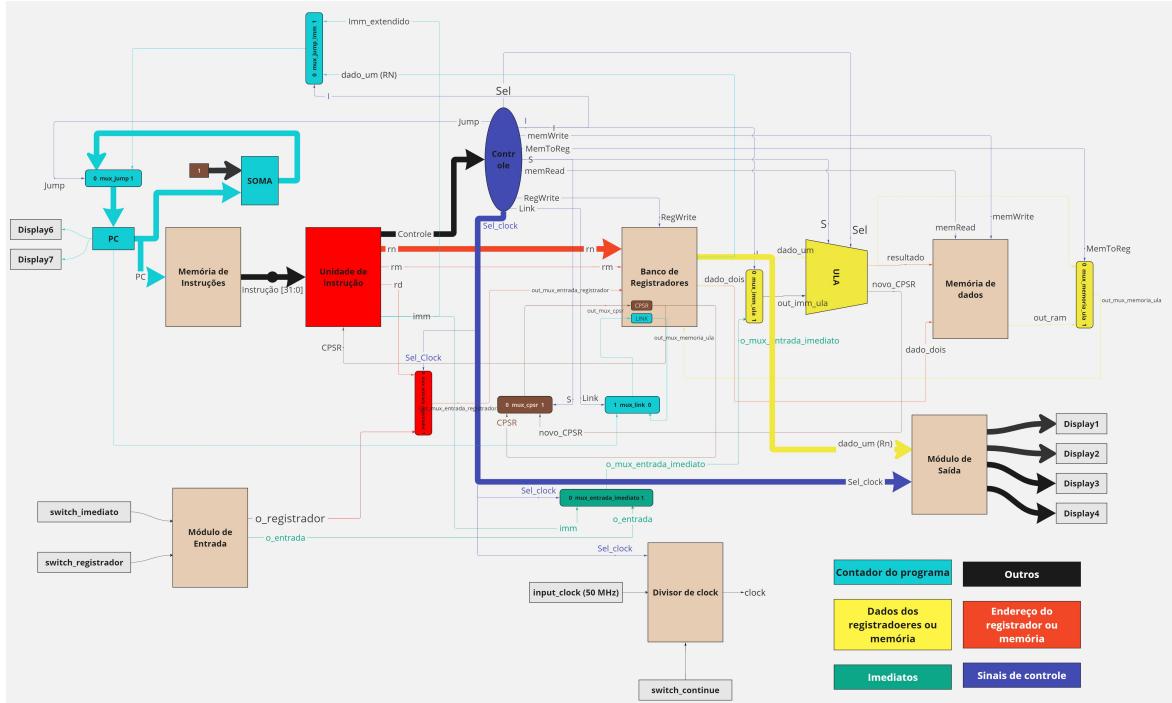
Fonte: O Autor (2024)

4.4.4 DataPath - Outras Instruções

As demais instruções possuem caminhos de dados bem distintos, por isso serão apresentadas as instruções de saída e entrada de dados.

A instrução de saída está ilustrada na figura 10, nela o pc é incrementado de maneira similar as outras instruções anteriores. A unidade de controle envia o sinal do registrador especial 29 para o módulo de saída, o qual atualiza os displays do FPGA caso o sinal de controle indique que seja uma instrução de saída.

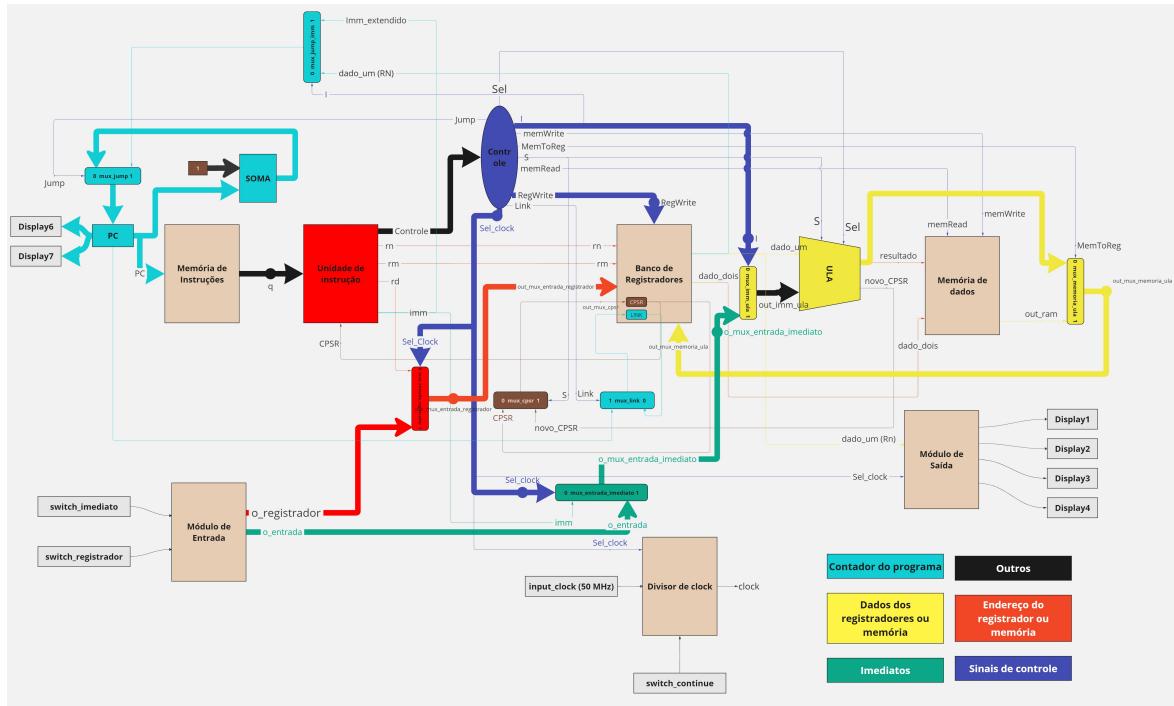
Figura 10 – Datapath das instrução OUT



Fonte: O Autor (2024)

Na instrução de entrada, o módulo de entrada recebe o valor do dado a ser salvo e o endereço de um registrador em que o dado será salvo. Para que esses sinais entrem no caminho de dados, a unidade de controle envia o sinal "sel clock" para os multiplexadores de entrada. O sinal segue um caminho similar a uma instrução de processamento de dados de soma e então é armazenada no registrador destino. Todo esse percurso pode ser visto na Figura 11.

Figura 11 – Datapath das instruções IN



Fonte: O Autor (2024)

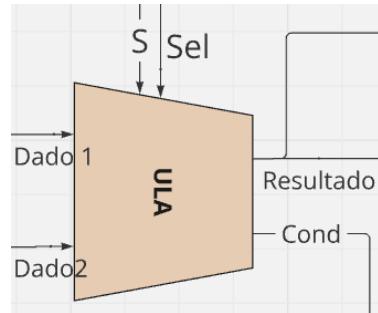
4.5 Implementação do Projeto

Com a utilização linguagem de descrição de hardware Verilog, cada unidade representada na Figura 6 foi implementada individualmente de modo a simular o seu funcionamento.

4.5.1 Unidade Lógica e Aritmética (ULA)

A Unidade Lógica e Aritmética (ULA) é responsável pela execução da maior parte das operações do processador, incluindo soma, subtração, multiplicação, divisão e operações lógicas. As entradas e saídas de dados da ULA são fixadas em 32 bits. Os sinais de controle são compostos por 4 bits para a seleção da operação e 1 bit para a ativação do envio de dados ao CPSR. Todas as entradas e saídas podem ser observados na figura 12. As operações executadas pela ULA podem ser examinadas na tabela 12.

Figura 12 – ULA



Fonte: O Autor (2024)

Tabela 12 – Operações da ULA e seus respectivos sinais de controle

Controle	Operação
0000	AND
0001	XOR
0010	OR
0011	SUB
0100	SUM
0101	MULT
0110	DIV

Fonte: O Autor (2024)

A implementação da ULA pode ser vista abaixo.

```

1 module ula (sel, a, b, s, resultado, cond);
2
3 input [3:0] sel;
4 input [31:0] a, b;
5 input s;
6 reg signed [31:0] res_int;
7                                     //Necessario para operacoes negativas
8 reg [3:0] cond_int;
9 output [31:0] resultado;
10 output [3:0] cond;
11
12 always @ (posedge sel or posedge a or posedge b or posedge s) begin
13     cond_int = 4'b0;
14     case(sel)
15         4'b0000 : res_int = a & b;
16                         //AND
17         4'b0001 : res_int = a ^ b;
18                         //XOR
19         4'b0010 : res_int = a | b;
20                         //OR
21         4'b0011 : begin
22                         res_int = a - b;
23                         if (s) begin
24                             if (res_int == 0) begin
25                                 cond_int = 4'b0010;
26                         end
27                         end
28                     end
29     endcase
30 end
31
32 endmodule

```

```

22                     end
23             begin
24
25                 if (res_int < 0) begin //NEGATIVE
26                     cond_int = 4'b0001;
27                 end
28                 else begin
29                     cond_int = 4'b0000; //POSITIVO
30                 end
31             end
32         end
33         4'b0100 : res_int = a + b; //SOMA
34         4'b0101 : res_int = a * b; //MULT
35         4'b0110 : res_int = a / b; //DIV
36         default : res_int = 32'b0;
37
38     endcase
39
40 end
41 assign resultado = res_int;
42 assign cond = cond_int;
43
44 endmodule

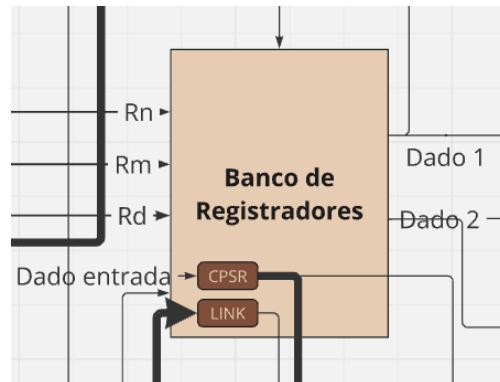
```

4.5.2 Banco de registradores

O banco de registradores consiste em um conjunto de 32 registradores de propósito geral. Esta implementação em Verilog utiliza um vetor, conforme o código abaixo. Cada registrador armazena 32 bits. As entradas para o banco de registradores incluem rn, rm, rd, a entrada de dados, o CPSR, o link e o *RegWrite*. As saídas são os dados 1 e 2, correspondentes aos valores armazenados em rn e rm, respectivamente. Todas as entradas e saídas estão representadas na Figura 13.

Além dos 32 registradores gerais, existem também os registradores especiais CPSR e link, cada um com suas próprias entradas e saídas. Para que o dado de entrada seja armazenado no registrador rd, o bit de controle chamado *RegWrite* deve estar ativo.

Figura 13 – Banco de registradores



Fonte: O Autor (2024)

```

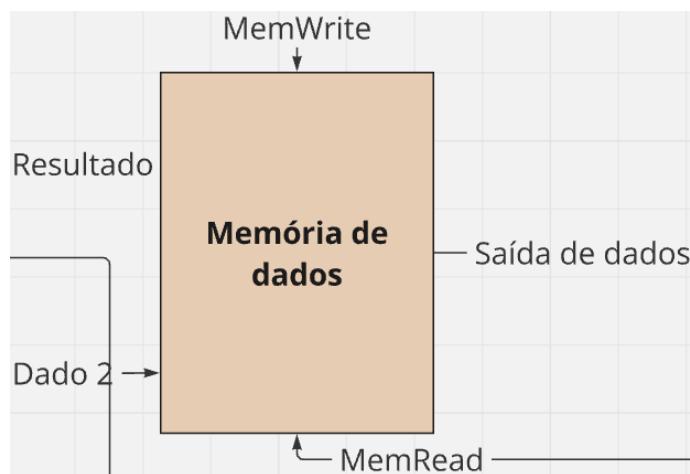
1 module b_registradores(
2     input wire [4:0] rn, rm, rd,
3     input wire reg_write, clock,
4     input wire [3:0] in_cpsr, in_link,
5     input wire [31:0] in_dados,
6     output reg [31:0] dado_um, dado_dois,
7     output reg [3:0] out_cpsr, out_link
8 );
9
10    reg [31:0] registradores [31:0];
11
12    // Bloco de inicializa o
13    integer i;
14    initial begin
15        for (i = 0; i < 32; i = i + 1) begin
16            registradores[i] = 32'b0; // Inicializa todos os registradores com 0
17        end
18    end
19
20    always @ (posedge clock) begin
21        if (reg_write) begin
22            registradores[rd] <= in_dados; // Atualiza o registrador apenas se reg_write
23                for 1
24            end
25            // Atualiza cpsr e link
26            registradores[30] <= {28'b0, in_cpsr}; // Ajusta o tamanho para 32 bits
27            registradores[31] <= {28'b0, in_link}; // Ajusta o tamanho para 32 bits
28        end
29
30        always @ (*) begin
31            dado_um = registradores[rn];
32            dado_dois = registradores[rm];
33            out_cpsr = registradores[30][3:0]; // Extrai os 4 bits de cpsr
34            out_link = registradores[31][3:0]; // Extrai os 4 bits de link
35        end
36
37    endmodule

```

4.5.3 Memória de dados (RAM)

A memória de dados, também conhecida como *random access memory* (RAM), é a memória principal do sistema. Ao contrário dos registradores, que são limitados a 32 unidades, a RAM possui um tamanho significativamente maior, com 1024 posições, cada uma com 32 bits. Entre as portas de entrada, estão o resultado da ULA e o dado 2, que pode ser um valor imediato ou um dado armazenado em um registrador, esse dado é utilizado para armazenamento na memória. Os bits de controle incluem memWrite, que é ativado para gravar dados na memória, e memRead, que é usado para ler dados da memória. A saída de dados é fornecida por uma porta de saída de 32 bits. Todas as portas estão ilustradas na figura 14.

Figura 14 – Memória de dados (RAM).



Fonte: O Autor (2024)

Para a implementação da memória de dados, foi utilizado um template fornecido pelo *Quartus Prime*. Foram ajustados apenas os parâmetros: DATA-WIDTH, configurado para 32, que define o tamanho dos espaços de memória, e ADDR-WIDTH, configurado para 10, que define a memória com 1024 posições. O template declara as entradas e saídas, define a variável RAM e implementa um bloco always para conectar o resultado acessado com a saída. Além disso, inclui uma estrutura condicional para implementar as funcionalidades de memRead e memWrite.

```

1 // Quartus Prime Verilog Temple
2 // Single part RAM with single read/write address
3
4 module memoria_dados
5 #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=10)
6 (
7     input [(DATA_WIDTH-1):0] data,
8     input [(DATA_WIDTH-1):0] addr,
9     input we, re, clk,
```

```

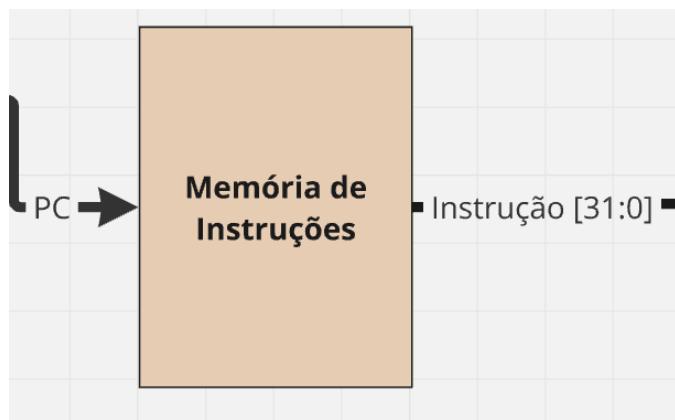
10      output reg [((DATA_WIDTH-1):0] q
11  );
12
13  // Declare the RAM variable
14  reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
15
16  always @ (posedge clk)
17  begin
18      // Write
19      if (we)
20          ram[addr] <= data;
21  end
22
23
24  // Countinuous assignment implies read returns NEW data.
25  // This is the natural behavior of the TriMatrix memory
26  // blocks in Single Port mode.
27
28  always @ (negedge clk)
29  begin
30      if (re)
31          q <= ram[addr];
32  end
33
34 endmodule

```

4.5.4 Memória de Instruções (ROM)

A memória de instruções (ROM) é a memória onde são armazenadas todas as instruções que serão lidas pelo processador, ela possui uma ligação direta com o PC, e uma saída que sai a instrução de 32 bits. Essa estrutura pode ser visualizada na figura 15.

Figura 15 – Memória de instruções (ROM).



Fonte: O Autor (2024)

Para sua implementação, foi utilizado um template fornecido pelo *Quartus Prime*. Os parâmetros do template foram ajustados da mesma forma que na memória de dados,

com a exceção do ADDR-WIDTH, que foi configurado para 5. Isso significa que o programa pode ter um total de 32 instruções. Um aspecto importante da implementação é a função readmemb, que permite a leitura de um arquivo de texto, no qual as instruções são armazenadas.

```

1 // Quartus Prime Verilog Template
2 // Single Port ROM
3
4 module memoria_instrucoes
5 #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=5)
6 (
7     input [(ADDR_WIDTH-1):0] addr,
8
9     output reg [(DATA_WIDTH-1):0] q
10 );
11
12 // Declare the ROM variable
13 reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];
14
15 // Initialize the ROM with $readmemb. Put the memory contents
16 // in the file single_port_rom_init.txt. Without this file,
17 // this design will not compile.
18
19 // See Section 17.2.8 for details on the
20 // format of this file, or see the "Using $readmemb and $readmemh"
21 // template later in this section.
22
23 initial
24 begin
25     $readmemb("single_port_rom_init.txt", rom);
26 end
27
28 always @ (*)
29 begin
30     q <= rom[addr];
31 end
32
33 endmodule

```

4.5.5 Módulo BCD

O módulo bcd converte um número decimal para uma string a qual será utilizada para enviar sinais para um display de sete segmentos.

```

1 module bcd(      input wire [4:0] entrada,
2                           output reg [6:0] display_7);
3
4     always @(*) begin
5         case(entrada)
6             4'b0000 : display_7 = 7'b1000000;
7             4'b0001 : display_7 = 7'b1111001;
8             4'b0010 : display_7 = 7'b0100100;
9             4'b0011 : display_7 = 7'b0110000;
10            4'b0100 : display_7 = 7'b0011001;
11            4'b0101 : display_7 = 7'b0010010;
12            4'b0110 : display_7 = 7'b0000010;
13            4'b0111 : display_7 = 7'b1111000;

```

```

14          4'b1000  : display_7 = 7'b0000000;
15          4'b1001  : display_7 = 7'b0011000;
16          default : display_7 = 7'bXXXXXXXX;
17      endcase
18  end
19 endmodule

```

4.5.6 Program Counter (PC)

O PC (Program Counter) é responsável por indicar qual instrução deve ser executada. Ele é controlado por um clock que determina o ritmo das mudanças. Além disso, o PC possui uma entrada para receber a nova linha de instruções a ser lida e uma saída, que é configurada pela função *assign*. Além da função principal, o pc também atualiza os displays 7 e 6 com o valor do pc atual. A implementação do PC pode ser vista abaixo.

```

1 module pc (    input  clock,
                  input wire [31:0] novo_estado,
                  output wire [31:0] o_pc,
                  output wire [7:0] display7, display6
5 );
6     wire [3:0] tens, units;
7     reg [31:0] pc;
8
9     always @ (posedge clock)
10    begin
11        pc <= novo_estado;
12    end
13
14    assign units = pc % 10;
15    assign tens = (pc % 100) / 10;
16
17    bcd bcd7(
18        .entrada(tens),
19        .display_7(display7)
20    );
21    bcd bcd6(
22        .entrada(units),
23        .display_7(display6)
24    );
25
26 assign o_pc = pc;
27 endmodule

```

4.5.7 Unidade de instrução

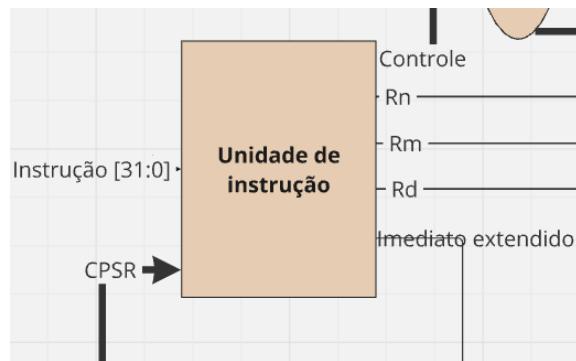
Esta unidade foi projetada para lidar com os diferentes tipos de instrução e suas variações de formato. Além disso, ela implementa a funcionalidade do registrador especial CPSR. Caso o campo cond da instrução não corresponda à informação contida no CPSR, a instrução a ser executada é a NOP, o que significa que nenhuma ação será realizada. Além dessas funcionalidades, a unidade também gera uma string de controle formatada,

que facilita o trabalho da unidade de controle. Essa string inclui todas as informações necessárias para ativar os multiplexadores e os bits de sinal para as demais unidades.

A unidade possui uma entrada para a instrução completa e outra para o valor atual do CPSR. Como saída, fornece os endereços de todos os registradores e o imediato, já estendido para 32 bits. Toda essa configuração pode ser visualizada na Figura 16.

Cabe ressaltar que apenas algumas condições do campo *cond* foram implementadas e que as demais devem ser implementadas no futuro.

Figura 16 – Unidade de instrução.



Fonte: O Autor (2024)

Para implementar diversas funcionalidades, foram necessárias várias instruções condicionais if e case. A primeira condição avalia se o campo *cond* corresponde ao CPSR, caso contrário, todas as saídas são configuradas para uma instrução NOP.

Para lidar com os diferentes formatos de instrução, foi utilizada uma estrutura case, onde cada tipo de instrução é tratado individualmente. Por fim, a saída de controle também é gerida de acordo com o tipo de instrução, sendo formatada segundo o padrão apresentado abaixo.

```

1 module unidade_instrucao (instrucao, in_cpsr, rn, rm, rd, imm, controle);
2     input wire [31:0] instrucao;
3     // Formato: [V, C, Z, N]
4     input wire [3:0] in_cpsr;
5
6     reg [4:0] r_rn, r_rm, r_rd;
7     reg [31:0] r_imm;
8     reg [10:0] r_controle;
9     reg verificacao;
10
11    output wire [4:0] rn, rm, rd;
12    output wire [31:0] imm;
13    output wire [10:0] controle;
14
15    always @ (*) begin
16        case(instrucao [31:28])
17            //EQ

```

```
18          4'b0000 : begin
19              if (in_cpsr[1]) begin
20                  verificacao = 1;
21              end
22              else begin
23                  verificacao = 0;
24              end
25          end
26
27          //NEQ
28          4'b0001 : begin
29              if (in_cpsr[1] == 0) begin
30                  verificacao = 1;
31              end
32              else begin
33                  verificacao = 0;
34              end
35
36          end
37          //CS
38          4'b0010 : begin
39      end
40          //CC
41          4'b0011 : begin
42      end
43
44          //MI
45          4'b0100 : begin
46              if (in_cpsr[0] == 1) begin
47                  verificacao = 1;
48              end
49              else begin
50                  verificacao = 0;
51              end
52
53          end
54
55
56          //PL
57          4'b0101 : begin
58              if (in_cpsr[0] == 0) begin
59                  verificacao = 1;
60              end
61              else begin
62                  verificacao = 0;
63              end
64
65          end
66          //VS
67          4'b0110 : begin
68      end
69          //VC
70          4'b0111 : begin
71      end
72          //HI
73          4'b1000 : begin
74      end
75          //LS
76          4'b1001 : begin
77      end
```

```
78          //GE
79          4'b1010    : begin
80            end
81          //LT
82          4'b1011    : begin
83            end
84          //GT
85          4'b1100    : begin
86            end
87          //LE
88          4'b1101    : begin
89            end
90          //ALLWAYS
91          4'b1110    : begin
92            verificacao = 1;
93            end
94
95        endcase
96
97
98
99        if (verificacao) begin
100          case({instrucao[27], instrucao[26]})  

101            // Processamento de dados
102            2'b00 : begin
103              r_rn = instrucao [19:15];
104              r_rd = instrucao [14:10];
105              // sel, I, S, L/s, L, U, Opcode
106              r_controle = {2'b00, instrucao[28], instrucao[20], 1'b0,
107                1'b0, 1'b0, instrucao[24:21]};
108
109            if (instrucao [25] == 1) begin // se a instru o
110              possuir imediato
111                r_imm = instrucao [9:0];
112                r_rm = 5'b0;
113              end
114            else begin
115              r_imm = 24'b0;
116              r_rm = instrucao [9:5];
117            end
118          end
119
120          //Transferencia de dados
121          2'b01 : begin
122            r_rn = instrucao [22:18];
123            if (instrucao [23] == 1) begin
124              r_rm = 5'b0;
125              r_rd = instrucao [17:13];
126
127            end
128            else begin
129              r_rm = instrucao [17:13];
130              r_rd = 5'b0;
131            end
132            r_imm = instrucao [12:0];
133            // sel, I, S, L/s, L, U, Opcode
134            r_controle = {2'b01, instrucao[25], 1'b0, instrucao[23],
135              1'b0, instrucao[24], 4'b0};
136          end
137        end
```

```

134
135
136          // Operações de desvio
137          2'b10 : begin
138              r_rd = 5'b0;
139              r_rm = 5'b0;
140              // sel, I, S, L/s, L, U, Opcode
141              r_controle = {2'b10, instrucao[25], 1'b0, 1'b0, instrucao
142                           [25], 1'b0, 4'b0};

143          if (instrucao [25] == 1) begin // se a instrução
144              possuir imediato
145                  r_imm = instrucao [23:0];
146                  r_rn = 5'b0;
147              end
148          else begin
149              r_imm = 24'b0;
150              r_rn = instrucao [23:19];
151          end
152      end

153
154          // Outras instruções
155          2'b11 : begin
156              // OUT
157              if (instrucao [25:23] == 3'b010)begin
158                  r_rn = 5'b11101;
159              end
160              else begin
161                  r_rn = 5'b0;
162              end
163              r_rm = 5'b0;
164              r_rd = 5'b0;
165              r_imm = 23'b0;
166              // sel, I, S, L/s, L, U, Opcode
167              r_controle = {2'b11, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, {1'b0,
168                           instrucao[25:23]}};

169          end
170      endcase
171  end
172  // Caso o CPSR seja diferente do cond a instrução executada ser a NOP
173  else begin
174      r_rn = 5'b0;
175      r_rm = 5'b0;
176      r_rd = 5'b0;
177      r_imm = 23'b0;
178      // sel, I, S, L/s, L, U, Opcode
179      r_controle = {2'b11, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 4'b0};
180  end
181
182  assign rn = r_rn;
183  assign rd = r_rd;
184  assign rm = r_rm;
185  assign imm = r_imm;
186  assign controle = r_controle;
187
188 endmodule

```

4.5.8 Multiplexadores

Os multiplexadores são essenciais para o funcionamento correto do processador. Eles são controlados pela unidade de controle e direcionam as informações através do caminho de dados. No total, são utilizados 6 multiplexadores, como ilustrado na Figura 6.

Todos os multiplexadores possuem uma estrutura semelhante, com 3 entradas, um seletor, dois dados e uma saída. As implementações em Verilog também seguem uma estrutura semelhante, utilizando uma única estrutura always com uma condicional para simular seu efeito. As implementações podem ser analisadas abaixo.

```

1 module mux_cpsr (in_s, in_atual, in_novo, out_mux_cpsr);
2     input wire      in_s;
3     input wire [3:0] in_atual, in_novo;
4
5     reg [3:0] r_out_cpsr;
6
7     output wire [3:0] out_mux_cpsr;
8
9     always @ (*) begin
10         if (in_s == 0) begin
11             r_out_cpsr = in_atual;
12         end
13         else begin
14             r_out_cpsr = in_novo;
15         end
16     end
17
18     assign out_mux_cpsr = r_out_cpsr;
19
20 endmodule

```

```

1 module mux_imm_ula (in_i, in_dado_dois, in_imm, out_imm_ula);
2     input wire      in_i;
3     input wire [31:0] in_dado_dois, in_imm;
4
5     reg [31:0] r_out_imm_ula;
6
7     output wire [31:0] out_imm_ula;
8
9     always @ (*) begin
10         if (in_i == 0) begin
11             r_out_imm_ula = in_dado_dois;
12         end
13         else begin
14             r_out_imm_ula = in_imm;
15         end
16     end
17
18     assign out_imm_ula = r_out_imm_ula;
19
20
21 endmodule

```

```

1 module mux_jump_imm (in_i, in_dado_um, in_imm, out_mux_jump_imm);
2     input wire      in_i;

```

```

3      input wire [31:0] in_dado_um, in_imm;
4
5      reg [31:0] r_out_dado_imm;
6
7      output wire [31:0] out_mux_jump_imm;
8
9      always @ (*) begin
10         if (in_i == 0) begin
11             r_out_dado_imm = in_dado_um;
12         end
13         else begin
14             r_out_dado_imm = in_imm;
15         end
16     end
17
18     assign out_mux_jump_imm = r_out_dado_imm;
19
20
21 endmodule

```

```

1 module mux_jump (in_jump, in_soma, in_imm, out_mux_jump);
2     input wire      in_jump;
3     input wire [31:0] in_soma, in_imm;
4
5     reg [31:0] r_out;
6
7     output wire [31:0] out_mux_jump;
8
9     always @ (*) begin
10        if (in_jump == 0) begin
11            r_out = in_soma;
12        end
13        else begin
14            r_out = in_imm;
15        end
16    end
17
18    assign out_mux_jump = r_out;
19
20
21 endmodule

```

```

1 module mux_link (in_link, in_pc, in_link_atual, mux_out_link);
2     input wire      in_link;
3     input wire [31:0] in_pc, in_link_atual;
4
5     reg [31:0] r_out_link;
6
7     output wire [31:0] mux_out_link;
8
9     always @ (*) begin
10        if (in_link == 0) begin
11            r_out_link = in_pc;
12        end
13        else begin
14            r_out_link = in_link_atual;
15        end
16    end
17
18    assign mux_out_link = r_out_link;

```

```
19
20
21 endmodule

1 module mux_memoria_ula (memtoreg, in_ula, in_memoria, out_mux_memoria_ula);
2     input wire      memtoreg;
3     input wire [31:0] in_ula, in_memoria;
4
5     reg [31:0] r_out_memoria_ula;
6
7     output wire [31:0] out_mux_memoria_ula;
8
9     always @ (*) begin
10         if (memtoreg == 0) begin
11             r_out_memoria_ula = in_ula;
12         end
13         else begin
14             r_out_memoria_ula = in_memoria;
15         end
16     end
17
18     assign out_mux_memoria_ula = r_out_memoria_ula;
19
20
21 endmodule
```

```
1 module mux_entrada_immediato(
2     input wire [31:0] dado_um, entrada,
3     input wire [1:0] sel_clock,
4     output wire [31:0] out_mux_entrada_immediato
5 );
6
7     reg [31:0] r_out_imm_entrada;
8
9
10    always @ (*) begin
11        if (sel_clock == 2'b10) begin
12            r_out_imm_entrada = entrada;
13        end
14        else begin
15            r_out_imm_entrada = dado_um;
16        end
17    end
18
19    assign out_mux_entrada_immediato = r_out_imm_entrada;
20
21
22 endmodule
```

```
1 module mux_entrada_registrador(
2     input wire [4:0] rd, entrada_registrador,
3     input wire [1:0] sel_clock,
4     output wire [4:0] out_mux_entrada_registrador
5 );
6
7     reg [4:0] r_out_mux_entrada_registrador;
8
9
10    always @ (*) begin
11        if (sel_clock == 2'b10) begin
```

```

12          r_out_mux_entrada_registrador = entrada_registrador;
13      end
14      else begin
15          r_out_mux_entrada_registrador = rd;
16      end
17  end
18
19 assign out_mux_entrada_registrador = r_out_mux_entrada_registrador;
20
21
22 endmodule

```

4.5.9 Unidade de controle

A unidade de controle é responsável por receber o sinal de controle da unidade de instrução e enviar os sinais apropriados para todos os multiplexadores, bem como para o funcionamento adequado de cada unidade do processador. Os sinais de controle são: Jump, MemWrite, MemRead, MemToReg, S, Link, RegWrite, I e selclock.

O sinal de controle é uma string de 11 bits que possui uma formatação especial ilustrada na tabela 13. Esta formatação é realizada pela unidade de instrução e ela facilita o processamento do sinal na unidade de controle. Caso um campo não esteja presente na instrução, ele será 0.

Tabela 13 – Formato do sinal de controle

[10:9]	[8]	[7]	[6]	[5]	[4]	[3:0]
Sel	I	S	L/S	L	U	Opcode

Fonte: O Autor (2024)

O sinal Jump determina se a próxima instrução será obtida a partir do endereço PC+1 ou de um valor arbitrário, especificado pelo código da instrução. O sinal MemWrite é direcionado para a Memória de Dados (RAM) e indica se o dado na entrada deve ser escrito, enquanto o sinal MemRead determina se a memória pode ser lida.

O sinal S define se uma instrução de processamento de dados pode modificar o valor do CPSR (Status do Programa), direcionando o resultado da condição (cond) da ULA para o registrador CPSR. Já o sinal Link realiza uma função similar ao sinal S, mas direciona o valor atual do PC para um registrador especial chamado Link.

O sinal RegWrite indica se o dado calculado pela ULA pode ser armazenado no Banco de Registradores. O sinal MemToReg seleciona qual informação será enviada para o Banco de Registradores, podendo ser o resultado calculado pela ULA ou o dado lido da Memória de Dados (RAM).

O sinal I indica se a instrução contém um valor imediato, direcionando-o diretamente para a ULA em instruções de processamento de dados ou para o PC em instruções de

desvio.

Além dos sinais de controle, a unidade de controle também envia uma string chamada "sel", que codifica a operação a ser executada pela ULA, e o sinal selclock, que informa o tipo de instrução em execução para a Unidade de Divisão do Clock e algumas outras unidades. Os possíveis valores para o selclock podem ser vistos na tabela 14, enquanto a implementação em verilog se encontra abaixo. O sinal de controle U para instruções LDR e STR ainda não foi implementado.

Tabela 14 – Operações do sinal SelClock

Instrução	SelClock
Normal	00
OUT	01
IN	10
Finish	11

Fonte: O Autor (2024)

```

1  module unidade_controle (
2      input wire [10:0] controle,
3      output reg      jump, memWrite, memRead, memToReg, s, i, regWrite, link,
4      output reg [1:0] sel_clock,
5      output reg [3:0] sel
6  );
7
8      always @ (*) begin
9          case (controle[10:9])
10             // instruções de processamento de dados
11             2'b00 : begin
12                 memWrite = 1'b0;
13                 memRead = 0;
14                 memToReg = 0;
15                 jump = 0;
16                 sel_clock = 2'b00;
17
18             case (controle[3:0])
19                 //AND
20                 4'b0001 : begin
21                     sel = 4'b0000;
22                     regWrite = 1;
23                 end
24
25                 //EOR
26                 4'b0010 : begin
27                     sel = 4'b0001;
28                     regWrite = 1;
29                 end
30
31                 //SUB
32                 4'b0011 : begin
33                     sel = 4'b0011;
34                     regWrite = 1;
35                 end
36

```

```
37          //ADD
38          4'b0101 : begin
39              sel = 4'b0100;
40              regWrite = 1;
41          end
42
43          //MRS
44          4'b0110 : begin
45              sel = 4'b0100;
46              regWrite = 1;
47          end
48
49          //MSR
50          4'b0111 : begin
51              sel = 4'b0100;
52              regWrite = 1;
53          end
54
55          //TST
56          4'b1000 : begin
57              sel = 4'b0000;
58              regWrite = 0;
59          end
60
61          //CMP
62          4'b1010 : begin
63              sel = 4'b0011;
64              regWrite = 0;
65          end
66
67          //ORR
68          4'b1100 : begin
69              sel = 4'b0010;
70              regWrite = 1;
71          end
72
73          //MOV
74          4'b1101 : begin
75              sel = 4'b0100;
76              regWrite = 1;
77          end
78
79          //MUL
80          4'b1110 : begin
81              sel = 4'b0101;
82              regWrite = 1;
83          end
84
85          //UDIV
86          4'b1111 : begin
87              sel = 4'b0110;
88              regWrite = 1;
89          end
90      endcase
91  end
92
93  //Transfer ncia de dados
94  2'b01 : begin
95      jump = 0;
96      sel = 4'b0100;
```

```
97                     sel_clock = 2'b00;
98
99                     // Instruções de load
100                    if (controle[6] == 1) begin
101                        memWrite = 0;
102                        memRead = 1;
103                        memToReg = 1;
104                        regWrite = 1;
105                    end
106
107                     // Instruções de store
108                    else begin
109                        memWrite = 1;
110                        memRead = 0;
111                        memToReg = 0;
112                        regWrite = 0;
113                    end
114                end
115
116                     // Instruções de desvio
117                2'b10 : begin
118                    jump = 1;
119                    sel = 4'b0100;
120                    memWrite = 0;
121                    memRead = 0;
122                    memToReg = 0;
123                    regWrite = 0;
124                    sel_clock = 2'b00;
125                end
126                     // OUTROS
127                2'b11 : begin
128                    case (controle[3:0])
129                        // NOP
130                        4'b0000 : begin
131                            jump = 0;
132                            sel = 4'b0100;
133                            memWrite = 0;
134                            memRead = 0;
135                            memToReg = 0;
136                            regWrite = 0;
137                            sel_clock = 2'b00;
138                        end
139                        // IN
140                        4'b0001 : begin
141                            jump = 0;
142                            sel = 4'b0100;
143                            memWrite = 0;
144                            memRead = 0;
145                            memToReg = 0;
146                            regWrite = 1;
147                            sel_clock = 2'b10;
148                        end
149                        // OUT
150                        4'b0010 : begin
151                            jump = 0;
152                            sel = 4'b0100;
153                            memWrite = 0;
154                            memRead = 0;
155                            memToReg = 0;
156                            regWrite = 0;
```

```

157                     sel_clock = 2'b01;
158                 end
159             //FINISH
160             4'b0011 : begin
161                 jump = 0;
162                 sel = 4'b0100;
163                 memWrite = 0;
164                 memRead = 0;
165                 memToReg = 0;
166                 regWrite = 0;
167                 sel_clock = 2'b11;
168             end
169         endcase
170     end
171 endcase
172
173     i = controle [8];
174     s = controle [7];
175     link = controle [5];
176
177 end
178 endmodule

```

4.5.10 Módulo de entrada

Como visto na figura 6, o módulo de entrada possui as entradas switchImediato e switchRegistrador, estas duas entradas são conectadas diretamente a interruptores do Kit FPGA, switchImediato recebe um dado, enquanto o switchRegistrador informa em qual registrador o dado será salvo, estas duas entradas são passadas diretamente para as saídas oEntrada e oRegistrador, respectivamente. A implementação deste unidade está logo abaixo.

Para que os valores da entrada sejam lidos apenas quando uma instrução IN é inserida, o processador faz uso de dois multiplexadores, um que recebe o endereço do registrador e o outro o dado a ser salvo.

```

1 module modulo_entrada
2 #(parameter TAM_ENTRADA=7)
3 (
4     input wire [TAM_ENTRADA:0] switch_imediato,
5     input wire [4:0] switch_registrador,
6     output reg [TAM_ENTRADA:0] o_entrada,
7     output reg [4:0] o_registrador
8 );
9
10
11 always @(*) begin
12     o_entrada = switch_imediato;
13     o_registrador = switch_registrador;
14 end
15
16
17 endmodule

```

4.5.11 Módulo de saída

O módulo de saída é a unidade que faz a integração entre o processador e os displays do Kit FPGA. Ele possui duas entradas, o *dadoUm* que vem do registrador 29 e um sinal de controle *selClock* que vem da unidade de controle, este sinal de controle gerencia a atualização dos displays. As saídas são os displays 1, 2, 3 e 4 do Kit FPGA, podendo mostrar números até 9999.

A implementação do módulo de saída foi realizada em duas partes, na primeira o número vindo do registrador 29 é separado em unidade, dezena, centena e milhar, estes valores são salvados em variáveis que posteriormente são convertidos em sinais para os displays de sete segmentos no Módulo bcd.

```

1 module modulo_saida(
2     input wire clock,
3     input wire [31:0] registrador_e,
4     input wire [1:0] sel_clock,
5     output wire [7:0] display1, display2, display3, display4
6 );
7     reg [4:0] d_display1, d_display2, d_display3, d_display4;
8
9     always @(negedge clock) begin
10         if(sel_clock == 2'b01)begin
11             d_display4 = registrador_e % 10;
12             d_display3 = (registrador_e % 100) / 10;
13             d_display2 = (registrador_e % 1000) / 100;
14             d_display1 = (registrador_e % 10000) / 1000;
15         end
16     end
17
18     bcd bcd1(
19         .entrada(d_display1),
20         .display_7(display1)
21     );
22     bcd bcd2(
23         .entrada(d_display2),
24         .display_7(display2)
25     );
26     bcd bcd3(
27         .entrada(d_display3),
28         .display_7(display3)
29     );
30     bcd bcd4(
31         .entrada(d_display4),
32         .display_7(display4)
33     );
34
35
36 endmodule

```

4.5.12 Módulo de divisão do clock

A unidade divisoria de clock reduz a frequência do clock para que as instruções possam ser visualizadas no Kit FPGA.

A primeira das entradas é a selClock, que seleciona o modo de operação da unidade de acordo com a instrução. Instruções normais são executadas em uma frequência mais alta, enquanto instruções de entrada e saída são executadas mais lentamente. A instrução finish trava o clock e sinaliza para o processador que o programa foi finalizado. A implementação dessa unidade pode ser vista abaixo.

O clock do FPGA é inserido pela entrada *CLOCK50* e, a cada quantidade estipulada de ciclos, o sinal é invertido. Isso permite gerar um sinal de clock com a frequência desejada para o processador. Com a frequência do clock do FPGA em 50 MHz, para selecionar uma frequência de 1 MHz, o sinal deve ser invertido a cada 25 pulsos. Para as instruções IN e OUT, o período escolhido foi de 5 segundos, então o sinal é invertido a cada 250 milhões de pulsos.

No código, a estrutura condicional da linha 40 impede que o clock continue a operar caso o sinal continue esteja em baixa. A instrução FINISH, codificada no sinal selClock, também atua como uma condição de parada do clock. Sua implementação pode ser vista na linha 34, onde o contador recebe continuamente o valor 0.

```

1 module div_clock
2 #(
3     parameter FREQ_CLOCK = 25, // Frequência do clock em Hz (metade)
4     parameter FREQ_CLOCK = 1, // Frequência do clock em Hz para testes
5     parameter PERIODO_OUT = 5, // Período do clock de saída para sel_clock = 01
6     parameter PERIODO_NORMAL = 1 // Período do clock normal para sel_clock = 00
7 )
8 (
9     input wire CLOCK_50,
10    input wire continue,
11    input wire [1:0] sel_clock,
12    output reg clk
13 );
14
15    reg [28:0] count;
16    reg [28:0] divisor;
17
18    initial begin
19        count = 0;
20        clk = 0;
21    end
22
23    always @ (posedge CLOCK_50) begin
24        // Definir o divisor com base no valor de sel_clock
25        case (sel_clock)
26            2'b00: divisor <= (FREQ_CLOCK * PERIODO_NORMAL); // NORMAL
27            2'b01: divisor <= (FREQ_CLOCK * PERIODO_OUT);      // OUT
28            2'b10: divisor <= (FREQ_CLOCK * PERIODO_OUT);      // IN
29            2'b11: divisor <= 0;                                // FINISH
30        default: divisor <= 0;
31    endcase
32
33    // Atualizar o contador e o sinal clk
34    if (sel_clock == 2'b11) begin
35        count <= 0;
36    end else if (count >= divisor) begin

```

```

37         clk <= ~clk;
38         count <= 0;
39     end else begin
40         if (sel_clock == 2'b10 && ~continue) begin
41             // N o incrementa count se 'continue' est  baixo
42         end else begin
43             count <= count + 1;
44         end
45     end
46 end
47 endmodule

```

4.5.13 Integração Final

Para unir todos os módulos foi criado um programa principal. Nesse programa as unidades são integradas através de *wires*, os quais estão nomeados na Figura 6. Além disto é nesse programa que os pinos de ligação com o FPGA são escolhidos.

```

1 module geral(
2     //FPGA
3     input wire input_clock,
4     //modulo_saida
5     output wire [6:0] display1, display2, display3, display4,
6
7     //modulo_entrada
8     input wire [7:0] switch_imediato, input wire [4:0] switch_registrador,
9     //div_clock
10    input wire switch_continue,
11    //output wire [31:0] saida_um, saida_dois, saida_tres,
12    //pc
13    output wire [7:0] display7, display6
14 );
15 //tester
16 //tester(
17 //    .entrada_um(resultado),
18 //    .entrada_dois(dado_dois),
19 //    .entrada_tres(dado_um),
20 //
21 //    .saida_um(saida_um),
22 //    .saida_dois(saida_dois),
23 //    .saida_tres(saida_tres)
24 //);
25
26
27 //conex es
28 //Modulo_entrada
29 wire [7:0] o_entrada;
30 wire [4:0] o_registrador;
31
32 //b_registradores
33 wire [31:0] dado_um, dado_dois;
34 wire [3:0] in_cpsr, out_cpsr;
35 wire [3:0] out_link;
36
37 //ula

```

```
38 wire [31:0] resultado;
39 wire [3:0] cond;
40
41 //mux_imm_ula
42 wire [31:0] out_imm_ula;
43
44 //mux_cpsr
45 wire [3:0] out_mux_cpsr;
46
47 //unidade_instrucao
48 wire [4:0] rn, rm, rd;
49 wire [31:0] imm;
50 wire [10:0] controle;
51
52 //memoria_instrucoes
53 wire [31:0] q;
54
55 //mux_link
56 wire [31:0] mux_out_link;
57
58 //pc
59 wire [31:0] o_pc;
60
61 //somador_pc
62 wire [31:0] out_somador_pc;
63
64 //mux_jump
65 wire [31:0] out_mux_jump;
66
67 //mux_jump_immm
68 wire [31:0] out_mux_jump_immm;
69
70 //memoria_dados
71 wire [31:0] out_ram;
72
73 //mux_memoria_ula
74 wire [31:0] out_mux_memoria_ula;
75
76 //mux_entrada_dado_um
77 wire [31:0] out_mux_entrada_immediato;
78
79 //mux_entrada_registrador
80 wire [31:0] out_mux_entrada_registrador;
81
82 //unidade de controle
83 wire [3:0] sel;
84 wire i, regWrite;
85 wire [1:0] sel_clock;
86
87
88
89 mux_entrada_registrador(
90     .rd(rd),
91     .entrada_registrador(o_registrador),
92     .sel_clock(sel_clock),
93     .out_mux_entrada_registrador(out_mux_entrada_registrador)
94 );
95
96 mux_entrada_immediato(
97     .dado_um(immm),
```

```
98     .entrada(o_entrada),
99     .sel_clock(sel_clock),
100    .out_mux_entrada_imediato(out_mux_entrada_imediato)
101 );
102
103 modulo_entrada(
104     .switch_imediato(switch_imediato),
105     .switch_registrador(switch_registrador),
106     .o_entrada(o_entrada),
107     .o_registrador(o_registrador)
108 );
109
110 modulo_saida(
111     .clock(clock),
112     .registrador_e(dado_um),
113     .sel_clock(sel_clock),
114     .display1(display4),
115     .display2(display3),
116     .display3(display2),
117     .display4(display1)
118 );
119
120 div_clock(
121     .CLOCK_50(input_clock),
122     .sel_clock(sel_clock),
123     .clk(clock),
124     .continue(switch_continue)
125 );
126
127
128 b_registradores br(      .rn(rn),
129     .rm(rm),
130     .rd(out_mux_entrada_registrador),
131     .in_dados(out_mux_memoria_ula),
132     .reg_write(regWrite),
133     .clock(clock),
134     .in_cpsr(out_mux_cpsr),
135     .in_link(mux_out_link),
136     .dado_um(dado_um),
137     .dado_dois(dado_dois),
138     .out_cpsr(out_cpsr),
139     .out_link(out_link)
140 );
141
142 ula u(
143     .sel(sel),
144     .a(dado_um),
145     .b(out_imm_ula),
146     .s(s),
147     .resultado(resultado),
148     .cond(cond)
149 );
150
151 mux_imm_ula(
152     .in_i(i),
153     .in_dado_dois(dado_dois),
154     .in_imm(out_mux_entrada_imediato),
155     .out_imm_ula(out_imm_ula)
156 );
157
```

```
158 mux_cpsr(
159     .in_s(s),
160     .in_atual(out_cpsr),
161     .in_novo(cond),
162     .out_mux_cpsr(out_mux_cpsr)
163 );
164
165 unidade_instrucao(
166     .instrucao(q),
167     .in_cpsr(out_cpsr),
168     .rn(rn),
169     .rm(rm),
170     .rd(rd),
171     .imm(imm),
172     .controle(controle)
173 );
174
175 memoria_instrucoes(
176     .addr(o_pc),
177     .q(q)
178 );
179
180 pc(
181     .clock(clock),
182     .novo_estado(out_mux_jump),
183     .o_pc(o_pc),
184     .display7(display7),
185     .display6(display6)
186 );
187
188 mux_link(
189     .in_link(out_link),
190     .in_pc(o_pc),
191     .in_link_atual(out_link),
192     .mux_out_link(mux_out_link)
193 );
194
195 somador_pc(
196     .in_pc(o_pc),
197     .out_somador_pc(out_somador_pc)
198 );
199
200 mux_jump(
201     .in_jump(jump),
202     .in_soma(out_somador_pc),
203     .in_imm(out_mux_jump_imm),
204     .out_mux_jump(out_mux_jump)
205 );
206
207
208 mux_jump_imm(
209     .in_i(i),
210     .in_dado_um(dado_um),
211     .in_imm(imm),
212     .out_mux_jump_imm(out_mux_jump_imm)
213 );
214
215 memoria_dados(
216     .data(dado_dois),
```

```
218     .addr(resultado),
219     .we(memWrite),
220     .re(memRead),
221     .clk(clock),
222     .q(out_ram)
223 );
224
225 mux_memoria_ula(
226     .memtoreg(memToReg),
227     .in_ula(resultado),
228     .in_memoria(out_ram),
229     .out_mux_memoria_ula(out_mux_memoria_ula)
230 );
231
232
233 unidade_controle(
234     .controle(controle),
235     .jump(jump),
236     .memWrite(memWrite),
237     .memRead(memRead),
238     .memToReg(memToReg),
239     .s(s),
240     .i(i),
241     .regWrite(regWrite),
242     .link(link),
243     .sel(sel),
244     .sel_clock(sel_clock)
245 );
246 endmodule
```

5 Resultados e Discussão

5.1 Forma de onda

Para verificar o funcionamento correto da implementação em Verilog, utiliza-se a técnica de simulação por forma de onda. Nesse processo, são introduzidas entradas para cada unidade com o objetivo de observar e analisar suas respostas.

5.1.1 PC

Na forma de onda da figura 17 é visto o sinal do pc sendo alterado conforme a entrada. Nota-se que a saída só é alterada quando o clock está na borda de subida.

Figura 17 – Forma de onda para o PC.

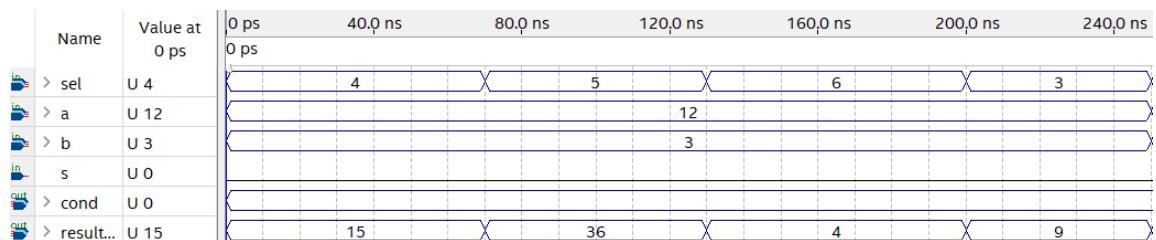


Fonte: O Autor (2024)

5.1.2 Unidade Lógica e Aritmética (ULA)

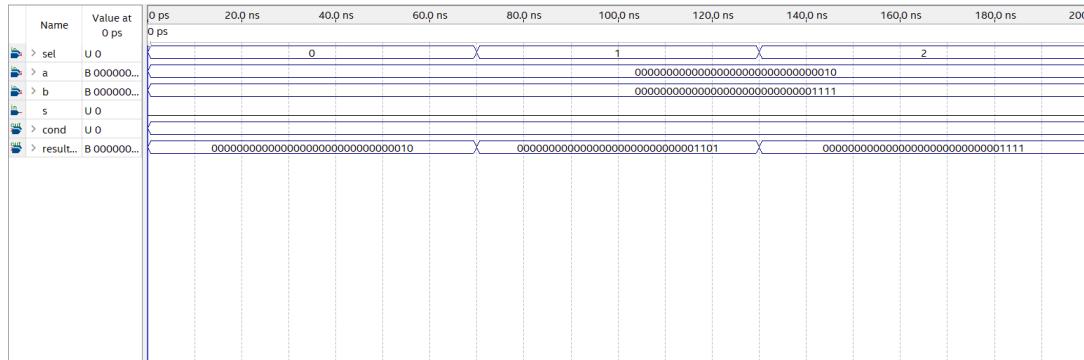
Abaixo, nas figuras 18 e 19, estão as formas de onda para as operações aritméticas e lógicas, respectivamente.

Figura 18 – Forma de onda para operações aritméticas na ULA.



Fonte: O Autor (2024)

Figura 19 – Forma de onda para operações lógicas na ULA.

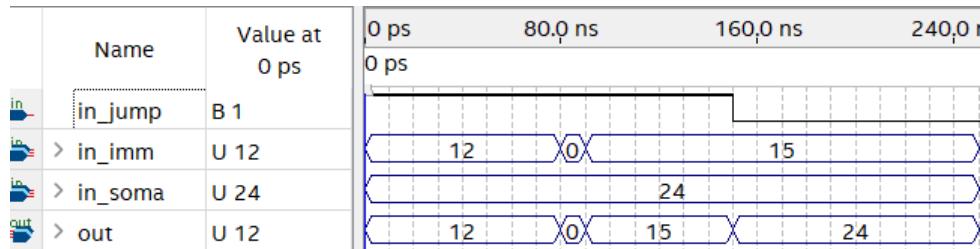


Fonte: O Autor (2024)

5.1.3 Multiplexadores

Abaixo, na figura 20, se encontra a forma de onda para o multiplexador do próximo estado do pc, ele é similar ao funcionamento dos demais.

Figura 20 – Forma de onda multiplexador PC.

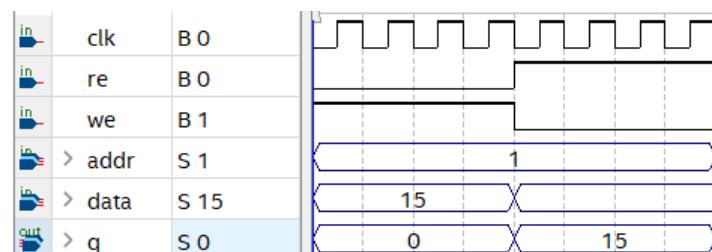


Fonte: O Autor (2024)

5.1.4 Memória RAM

Na forma de onda apresentada na Figura 21, o valor 15 é armazenado no endereço 1 da memória e logo em seguida é lido. Essas operações só são possíveis graças aos sinais *read* e *write* que estão em alta quando cada operação é realizada.

Figura 21 – Forma de onda memória RAM.

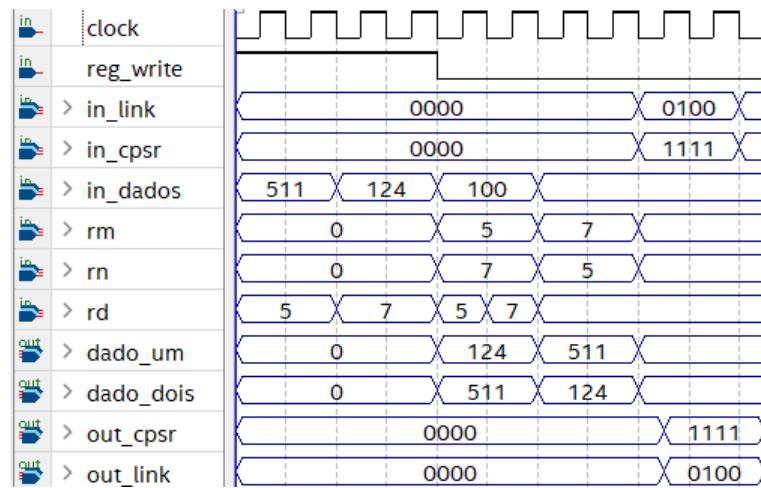


Fonte: O Autor (2024)

5.1.5 Banco de registradores

Na forma de onda apresentada na Figura 22, o banco de registradores é testado. Inicialmente, os valores 511 e 124 são armazenados nos registradores 5 e 7, respectivamente. Esses valores são lidos pelas entradas rm e rn. Em seguida, os registradores especiais são testados pelas entradas inlink e incpsr. É importante ressaltar que os valores são salvos nos registradores apenas quando o bit regWrite está ativo, como mostrado quando o valor 100 é inserido em rd, mas não é salvo.

Figura 22 – Forma de onda do banco de registradores.



Fonte: O Autor (2024)

5.1.6 Memória de instruções (ROM)

A memória de instruções recebe as instruções de um arquivo .txt e transforma cada uma das linhas em uma instrução diferente, sendo a primeira com endereço 0. Como exemplo, as instruções mostradas na figura 23 foram utilizadas.

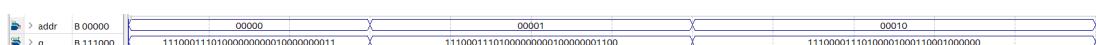
A figura 24 mostra que cada instrução foi alocada de maneira correta.

Figura 23 – Arquivo .txt de entrada

1	11100011101000000000010000000011
2	11100011101000000000100000001100
3	11100001110100001000110001000000
4	

Fonte: O Autor (2024)

Figura 24 – Forma de onda memória ROM



Fonte: O Autor (2024)

5.1.7 Unidade de Instrução

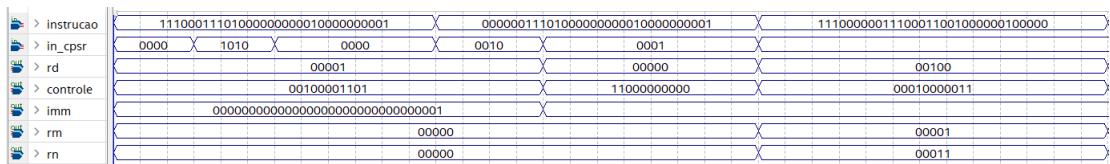
Para a unidade de instrução foram realizados três testes descritos abaixo e ilustrados na Forma de onda da figura 25.

A primeira instrução, "MOVI Rd = 1, Imm = 1", é executada de forma independente do estado do CPSR, uma vez que seu Campo COND é igual a AL (Always).

A segunda instrução, "MOVIEQ Rd = 1, Imm = 1", é executada somente quando o bit de zero (Z) é igual a 0. Caso contrário, a instrução em questão não será executada, um valor zero será enviado para todas as portas e o código da instrução NOP será enviado para o unidade de controle.

Por fim, a terceira instrução, "SUBS rn = 3, rm = 1, rd = 4", utiliza todos os registradores, mas não inclui valores imediatos.

Figura 25 – Forma de onda para a unidade de instrução.

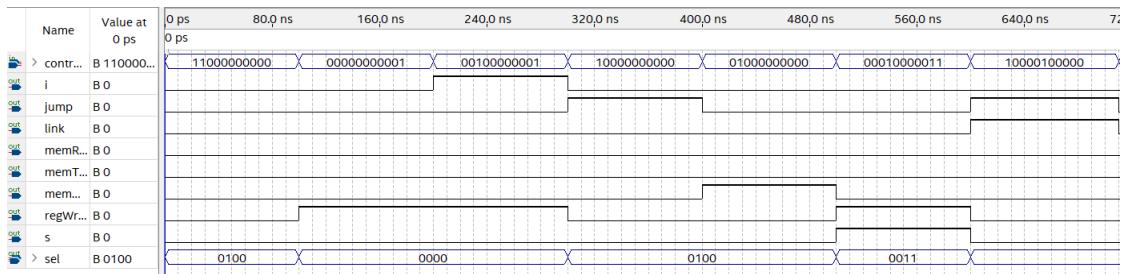


Fonte: O Autor (2024)

5.1.8 Unidade de controle

Na forma de onde apresentada na figura 26 foram testadas as instruções NOP, AND, ANDI, B, STR, SUBS e BL respectivamente e como pode ser percebido, todas as instruções foram corretamente decodificadas.

Figura 26 – Forma de onda da Unidade de Controle.



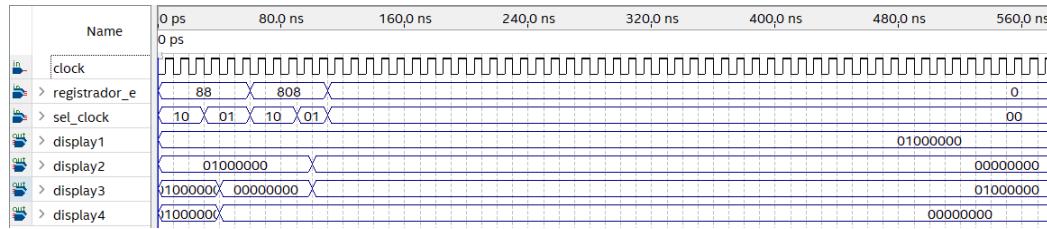
Fonte: O Autor (2024)

5.1.9 Módulo de saída

No teste do módulo de saída, foi inserido o valor 88 no registrador de entrada e, como visto na forma de onda, os displays se alteraram apenas quando o selClock é 01. Em

seguida observa-se o mesmo comportamento para o número 808 na figura 27.

Figura 27 – Forma de onda para o módulo de saída

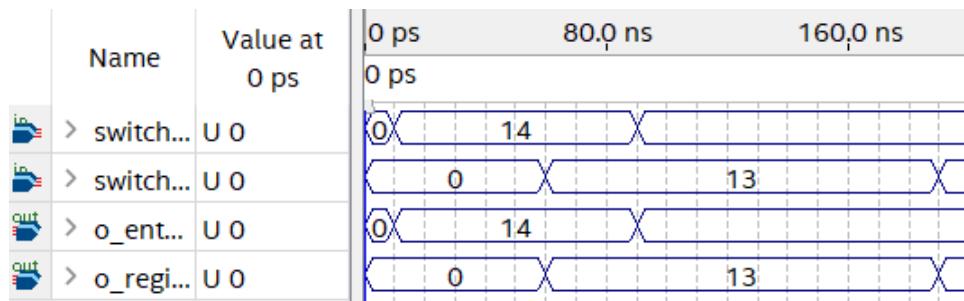


Fonte: O Autor (2024)

5.1.10 Módulo de entrada

O teste apresentado na figura 28 possui um resultado mais simples de ser interpretado, o valor da saída vai ser igual ao valor do switch de entrada.

Figura 28 – Forma de onda para o módulo de entrada

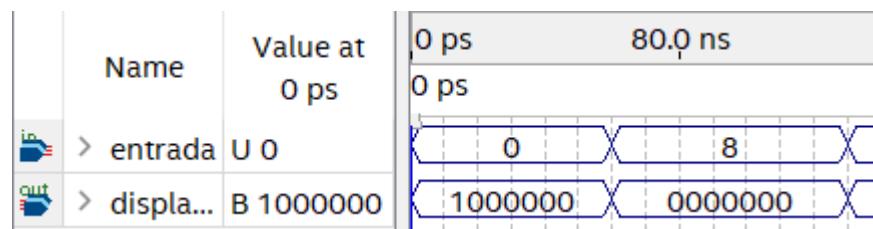


Fonte: O Autor (2024)

5.1.11 Módulo BCD

O módulo bcd também apresenta o comportamento esperado, o valor da entrada é codificado para a saída. Na figura 29 os valores codificados são: 0 e 9.

Figura 29 – Forma de onda para o módulo bcd



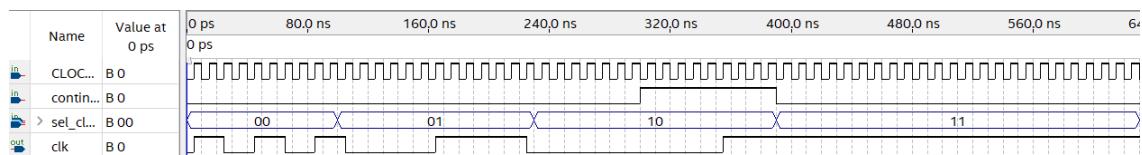
Fonte: O Autor (2024)

5.1.12 Unidade de divisão do clock

A unidade de divisão do clock apresenta o comportamento esperado. O clock é inserido na entrada padrão. Quando o sinal selClock é 00, o clock opera com uma frequência específica. Se selClock é alterado para 01, a frequência diminui. Quando selClock é 10, o clock começa a contar somente quando o sinal continue está em alta. Finalmente, sinais 11 em selClock interrompem o clock indefinidamente.

Vale ressaltar que, para o teste, os parâmetros do período final da divisão do clock foram ajustados para melhor visualização na forma de onda apresentada na Figura 30.

Figura 30 – Forma de onda para unidade de divisão do clock



Fonte: O Autor (2024)

5.1.13 Programas

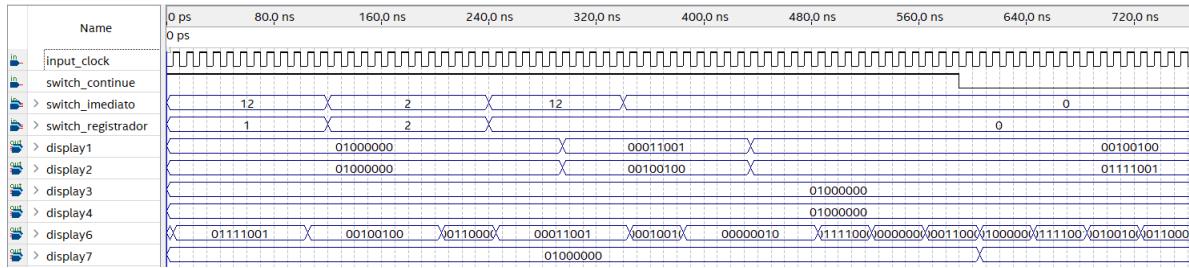
5.1.13.1 Área do triângulo

O programa em assembly a seguir recebe o valor da base e da altura de um triângulo e em seguida mostra no display o resultado da multiplicação seguido pelo valor da área. Na figura 31 o programa é visto em execução.

Primeiro o número 12 e 2 são inseridos no *switch imediato*, nos registradores 1 e 2, respectivamente. Em seguida percebe-se que o display 1 muda de 0 para 4 e o display 2 de 0 para 2, representando o número 24. Em seguida eles alteram novamente para o número 12, o que corresponde ao planejado.

Já os displays 6 e 7 mostram o valor da unidade e dezena do pc, respectivamente.

Figura 31 – Forma de onda para o programa da área de um triângulo



Fonte: O Autor (2024)

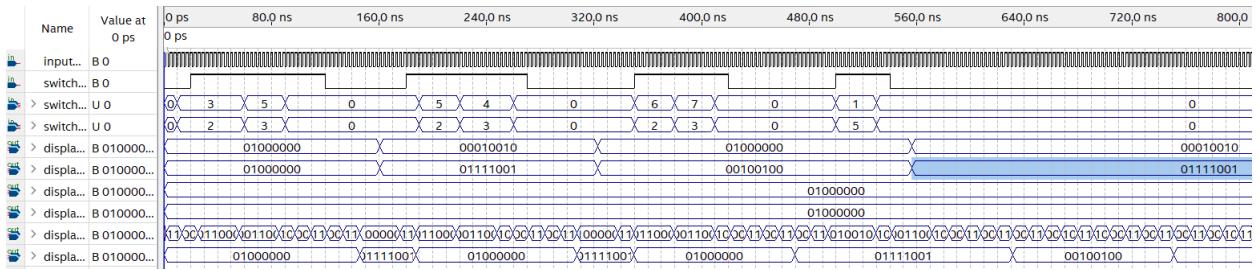
5.1.13.2 Área de retângulos

Para testar todas as funcionalidades do processador, foi desenvolvido um programa que utiliza todos os tipos de instrução para calcular várias áreas de retângulos até que uma das áreas atinja o valor estipulado. Quando essa condição é satisfeita, o programa solicita outro número inteiro n e exibe, nos displays, a área do retângulo na posição n-ésima.

O programa inicia com uma instrução NOP, seguida do início do contador do registrador 1 na posição 1. Na linha 3, é definido o valor máximo para a área dos retângulos. Em seguida, são lidos os valores da base e da altura. Na linha 7, a instrução CMP subtrai a área calculada do valor máximo armazenado em R4. Se a operação resultar em um número negativo, o CPSR será configurado para 0010, e as próximas instruções serão executadas. Caso contrário, as próximas instruções não serão executadas, e o programa solicitará a posição a ser exibida no display. Por fim, o programa é encerrado.

Todas essas instruções podem ser observadas na forma de onda da figura 32. A sequência de bases e alturas inseridos foram (3,5), (5,4) e (6,7) para sair do loop. Todos as áreas menores que 30 são exibidas nos displays, assim como a área do primeiro triângulo ao final da execução.

Figura 32 – Forma de onda para o programa: Área de retângulos



Fonte: O Autor (2024)

Para compilar o último programa, foi necessário quase 30 minutos, como mostrado na Figura 33. Uma possível justificativa para esse tempo prolongado é a má otimização do código, como indicado pela mensagem exibida durante a compilação: *"Info (188005): Design requires adding a large amount of routing delay for some signals to meet hold time requirements"*. Essa mensagem sugere que há dificuldades em unir os sinais dos módulos no programa principal.

Figura 33 – Tempo de compilação do programa: Área de retângulos

	Task	Time
✓	▼ ➤ ...	00:29:59
✓	➤ ➤	00:00:18
✓	➤ ➤	00:27:17
✓	➤ ➤	00:00:04
✓	▼ ➤	00:02:20

Fonte: O Autor (2024)

6 Considerações finais

Ao final da disciplina, a implementação do processador ARM foi um sucesso. O processador conseguiu realizar os quatro tipos de operações descritas: processamento de dados, transferência de dados, instruções de desvio e operações especiais, conforme demonstrado no último programa apresentado. Além disso, o processador foi sintetizado no FPGA com êxito, evidenciando que, apesar da alta frequência do clock, o módulo de divisão de clock, em conjunto com os módulos de entrada e saída, se mostrou eficaz.

Apesar de todos os objetivos terem sido cumpridos, ainda há melhorias e recursos que podem ser adicionados no futuro.

Dentre os pontos de melhoria, destaca-se a otimização do código, que se revelou necessária no último programa a ser compilado. Esse programa demorou quase meia hora para ser compilado, o que tornaria inviável a compilação de programas ainda mais complexos.

No final, foi implementada apenas uma operação que modifica o CPSR: a subtração. No entanto, outras operações, como ADDS e ORRS, também podem alterar o CPSR, ajustando os bits C e V que não foram utilizados. Uma segunda operação ainda não implementada é o bit U das instruções LDR e STR. Embora essas operações não tenham sido implementadas, todos os espaços no código da unidade de instrução e na ULA estão devidamente descritos, o que facilita uma possível implementação futura.

Referências

Altera Corporation. *DE2-115 User Manual*. [S.l.], 2013. DE2115-UM-01. Disponível em: <https://www.terasic.com.tw/attachment/archive/502/DE2_115_User_manual.pdf>. Citado na página 17.

ARM. *ARM7TDMI-S Data Sheet*. [S.l.], 2000. ARM DDI 0084D. Disponível em: <<https://developer.arm.com/documentation/ddi0084/latest>>. Citado 2 vezes nas páginas 13 e 15.

ARM Limited. *ARM® Architecture Reference Manual*. [S.l.], 2008. DDI 0406B. Disponível em: <<https://documentation-service.arm.com/static/5f8dacfff86e16515cdb8bb1>>. Citado na página 12.

Arm Limited. *Arm® Architecture Registers for A-profile architecture*. [S.l.], 2024. DDI 0601 (ID032524). Disponível em: <<https://developer.arm.com/documentation/ddi0601/2024-03/AArch32-Registers/CPSR--Current-Program-Status-Register>>. Citado na página 15.

IFSC. *Arquivo:Fig050 MCO018703.png*. 2020. Acessado em: 19 de abril de 2024. Disponível em: <https://wiki.sj.ifsc.edu.br/index.php/Arquivo:Fig050_MCO018703.png#filelinks>. Citado na página 12.

STALLINGS, W. *Arquitetura e Organização de Computadores 8a Edição*. [S.l.]: São Paulo: Prentice Hall do Brasil, 2010. Citado 4 vezes nas páginas 7, 8, 10 e 11.