RA: 133811

DESENVOLVIMENTO DE UM SISTEMA COMPUTACIONAL EM LÓGICA PROGRAMÁVEL BASEADO EM MIPS MONOCICLO

São José dos Campos - Brasil

Outubro de 2020

RA: 133811

DESENVOLVIMENTO DE UM SISTEMA COMPUTACIONAL EM LÓGICA PROGRAMÁVEL BASEADO EM MIPS MONOCICLO

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

Outubro de 2020

Resumo

A importância de se estudar e desenvolver sistemas computacionais provém do quão dependente a humanidade está e se manterá em relação à tecnologia. Em suma, o objetivo deste projeto foi desenvolver um processador funcional do zero em lógica programável Verilog no *software* Quartus Prime utilizando um kit FPGA para interação E/S. Os resultados esperados, individualmente e coletivamente, se apresentaram consistes e corretos de acordo com as propostas pre-instauradas, sendo utilizadas simulações de casos generalizados nos compomentes implementados e, uma vez concatenados, um algoritmo de Fibonacci para testar o processador. Comprovado que o processador individualmente e coletivamente funcionava, houve a confirmação do sucesso em desenvolver tal projeto apesar das dificuldades encontradas.

Palavras-chaves: Sistema computacional. RISC. Harvard. Monociclo. MIPS. 32 bits. FPGA. Quartus Prime. Verilog.

Lista de ilustrações

Sumário

1 Introdução

Pensar em evolução sem tecnologia é impossível. Ambos conceitos estão intrinsecamente interligados de maneira inseparável, toda evolução dependeu de tecnologia à certo nível, como temos em diversos exemplos no decorrer da história: manejo do fogo, ferramentas de pedra polida, confecção de ferramentas a partir de metais, pólvora, industrialização, etc. E, mesmo que talvez a humanidade se considere no ápice de seu conhecimento, não estamos livre da evolução e, consequentemente, da tecnologia no geral, mas fica a dúvida: qual é o passo seguinte para evoluirmos?

Segundo Hennessy (1), "...computação fez um progresso incrível no decorrer dos últimos 65 anos, desde que foi criado o primeiro computador eletrônico de uso geral. Hoje, por menos de US$ 500 se compra um computador pessoal com mais desempenho, mais memória principal e mais armazenamento em disco do que um computador comprado em 1985 por US$ 1 milhão. Essa melhoria rápida vem tanto dos avanços na tecnologia usada para montar computadores quanto da inovação no projeto de computadores".

Dada tal citação, podemos concluir que não se trata mais de fogo ou pólvora como no passado para evoluirmos: atualmente todas áreas do conhecimento que utilizamos à nosso favor se encontram armazenadas e disponibilizadas na grande *world wide web*, acessadas a partir de um celular ou um computador que a grande maioria das pessoas tem possibilidade de ter graças ao progresso computacional. As antigas ferramentas de pedra, hoje são os sistemas computacionais que, graças a projetos e estudos na áreas de computação, como por exemplo, na área de arquitetura e organização de computadores, se desenvolvem todos os dias propiciando mais acessibilidade e tecnologia, portanto, evolução.

2 Objetivos

2.1 Geral

Desenvolvimento de um sistema computacional, composto por processador, memória e interface de comunicação E/S, baseado na arquitetura MIPS monociclo, em lógica programável.

2.2 Específico

O desenvolvimento do sistema computacional e seus módulos puderam ser descritos e definidos, especificamente, nas seguintes etapas:

1. Arquitetura Base e Conjunto de Instruções:
    - Determinação da arquitetura base utilizada como modelo;
    - Elaboração do formato de instrução de acordo com a arquitetura base e palavra;
    - definida;
    - Elaboração dos modos de endereçamento das instruções planejadas;
    - Elaboração e categorização do conjunto de instruções utilizados;
    - Elaboração e esquematização da arquitetura base utilizada.
2. Implementação dos componentes em Verilog e simulações:
    - Implementação do PC;
    - Implementação da Memória de Instruções;
    - Implementação da Memória de Dados;
    - Implementação do Extensor de 16 para 32 bits;
    - Implementação do Extensor de 26 para 32 bits;
    - Implementação do Multiplexador do PC;
    - Implementação do Multiplexador do Registrador de Escrita;
    - Implementação do Multiplexador do Dado para a ULA;
    - Implementação do Multiplexador do Dado de Escrita;
    - Implementação do Banco de Registradores;
    - Implementação da Interface de E/S;

- Implementação da ULA;
- Implementação da UC;
- Simulação individual de cada módulo;
- Concatenação de todos os módulos implementados;
- Simulação geral do sistema computacional.

3 Fundamentação Teórica

Resgatando elementos cruciais para o desenvolvimento do projeto, esta seção existe em função de trazer o necessário conceitualmente falando para o entendimento e própria implementação do projeto. Trazendo do conceito mais básico ao mais complexo utilizado resumidamente.

3.1 Sistema Computacional

Segundo Stallings (2), "Um sistema computacional, como qualquer outro sistema, consiste em um conjunto de componentes interrelacionados. O sistema é mais bem caracterizado em termos de estrutura (o modo como os componentes são interconectados) e função (a operação dos componentes individuais).", sendo tais componentes descritos, ainda segundo Stallings (2), "Sistema computacional: seus principais componentes são processador, memória e E/S".

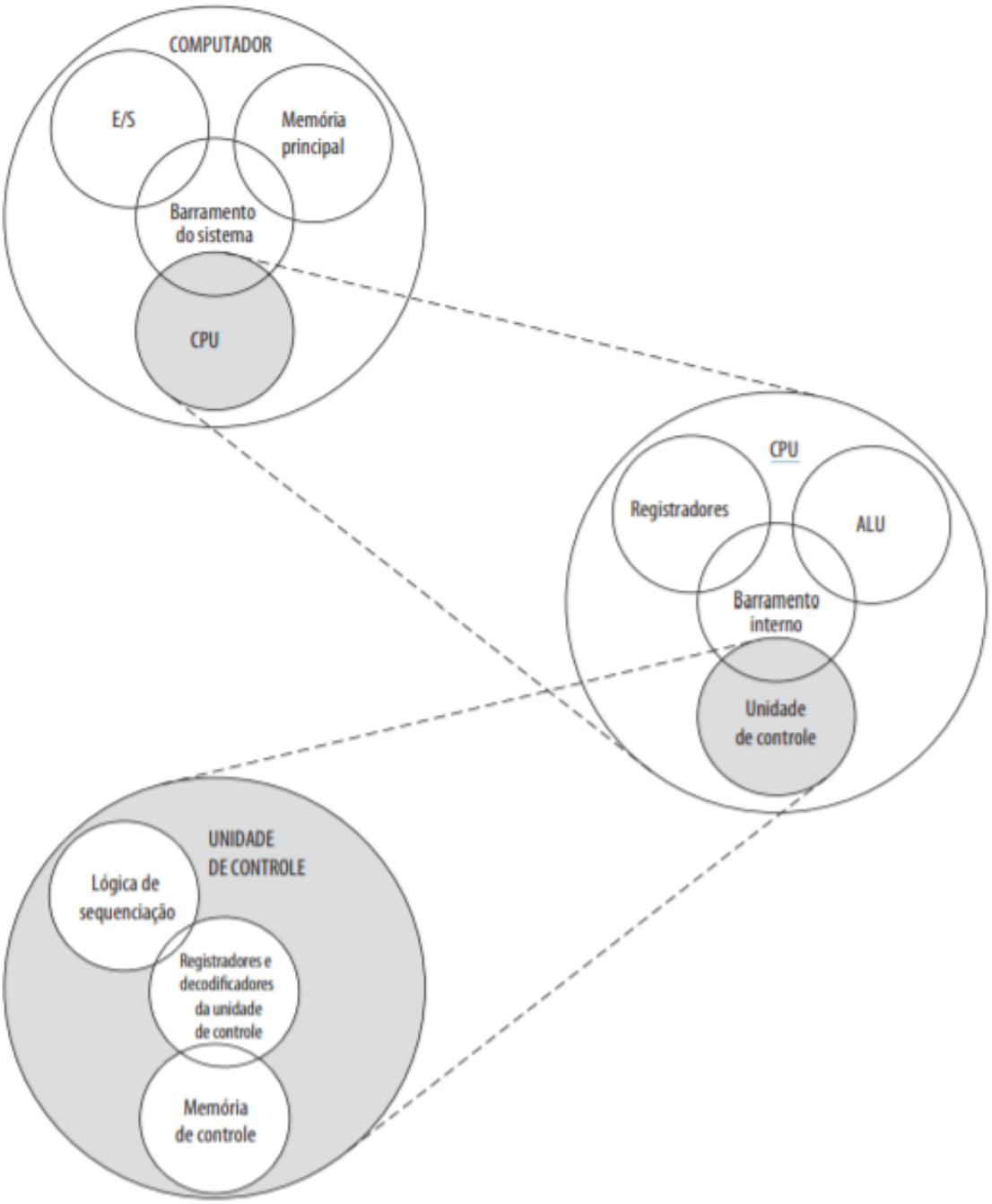
O processador é também conhecido como CPU, acrônimo de Central *Processing Unit*, do inglês, Unidade Central de Processamento. Consiste em três módulos: a ULA, Unidade Lógica e Aritmética, cuja função é resolver toda manipulação lógica e aritmética necessária de dados; a Unidade de Controle, cuja função é controlar os pontos de execução, os desvios, decodificar as instruções e buscar os operandos necessitados por estas; e um banco de registradores, variáveis existentes determinadas de acordo com a arquitetura definida que armazenam dados para o processamento.

A memória, podendo ser dividida em memória de instruções e memória de dados, são quantidades limitadas de bits cuja função é armazenar, no caso do primeiro tipo de memória: instruções, e no segundo, dados.

O termo "E/S" se refere às interfaces de entrada/saída do sistema computacional, resumidamente, a forma que o computador se utiliza para gerar entradas e saídas através dos periféricos conectados à esse.

A Figura 1 representa a um sistema computacional mínimo e seus módulos, de acordo com Stallings:

Figura 1 – Sistema Computacional



Fonte: Arquitetura e Organização de Computadores (2)

3.2 RISC X CISC

Quando se pretende construir um sistema computacional é comum se deparar com o conceito RISC e CISC independente do lugar que você utiliza para pesquisar sobre, porém o que são tais conceitos e por que eles se contrapõem?

Segundo Stallings (2), “Durante muitos anos, a tendência geral na arquitetura e

organização de computadores foi aumentar a complexidade do processador: mais instruções, mais modos de endereçamento, mais registradores especializados e assim por diante.", tal constatação é a orientação direta do que o conceito CISC, acrônimo do inglês para *Complex Instruction Set Computer*, ou, em uma tradução literal, "Computador com um Conjunto Complexo de Instruções" se trata: resumidamente um tipo sistema computacional que trás consigo um conjunto de instruções contendo diversas especificações para manipulação dos dados, instruções complexas que visam consequentemente gerar arquiteturas mais complexas e simplificar a construção de algoritmos estas.

Em contrapartida, RISC, acrônimo de *Reduced Instruction Set Computer*, ou, em uma tradução literal, "Computador com um conjunto reduzido de instruções", se trata de um tipo de sistema computacional que trás consigo um conjunto de instruções simples e bem definidas, cuja concatenação as tornam mais complexas e específicas caso surja a necessidade, cabendo ao programador manipulá-las de modo que o satisfaça.

3.3 Arquitetura de Harvard

Havendo necessidade de pôr o microcontrolador a trabalhar mais rápido nos sistemas computacionais em geral e indo em na direção oposta da arquitetura de Von Neumann (definida pela característica de armazenar instruções e dados no mesmo espaço de memória), a arquitetura de Harvard é caracterizada pela cisão da memória de instruções e da memória de dados, permitindo o acesso simultâneo a ambas e acelerando o processo de execução e procura de instruções, uma vez que em um mesmo clock há a possibilidade de submeter uma instrução nova enquanto se realiza a anterior.

Comumente encontrada em arquiteturas RISC, tal arquitetura auxilia também na redução do conjunto de instruções, o que acaba unindo-as perfeitamente, uma vez que arquiteturas RISC tendem a precisar acelerar o processos dos quais são submetidas, já que a junção de instruções menores resultam em maiores quantidades de ciclos para obtenção do que foi submetido.

3.4 Conjunto de Instruções

Uma vez construído o "cérebro e corpo" do sistema computacional, o que resta é dar funcionalidade para este através de ordens, ou melhor, instruções.

Conjunto de instruções de uma arquitetura, ou do inglês ISA, acrônimo para *Instruction Set Architecture*, são as possíveis operações que o sistema consegue realizar utilizando dados através de seus módulos físicos a fim de realizar algo, por exemplo, calcular a soma de dois números ou até mesmo calcular Fibonacci. Normalmente categorizados em funções aritméticas, lógicas, condicionais ou de controle, quem decide e monta as instruções

é o projetista da máquina, quem as utiliza é o programador para montar softwares na máquina projetada.

Um limite onde o projetista de computador e o programador de computador podem ver a mesma máquina é o conjunto de instruções de máquina. (STALLINGS, 2010) (2).

O conjunto de instruções é projetado através de tipos, ou formatos, feitos para serem compreendidos exclusivamente pelo sistema computacional, portanto o projetista deve se atentar em quais formatos pretende utilizar, em seu tamanho em bits (palavra), nos tipos de dados a serem processados naquele sistema e de que forma. Tais formatos normalmente seguem um padrão em sua construção uma vez que necessitam de partes primordiais para funcionar:

1. Opcode: acrônimo para *Operation Code*, do inglês, código de operação, é um código normalmente binário subentendido pela Unidade de Controle do sistema operacional, que indica o *datapath* que a instrução deve passar com os dados;

2. Operando de leitura: é um dado que será utilizado normalmente pela ULA, pode ser uma constante dada diretamente (conhecido como imediato), ou uma constante armazenada no banco de registradores ou na memória de dados;

3. Operando de escrita: é um dado resultante das operações realizadas pelas instruções dadas, armazenado no banco de registradores ou na memória de dados, pode ser utilizado mais tarde como um operando de leitura;

4. Endereço: imediato ou dado passado pelas memórias, tem função de passar um valor específico para o PC, acrônimo para *Program Counter*, do inglês, Contador de Programa (registrador específico responsável pela chamada das instruções), a fim de fazer um desvio nas instruções.

Por exemplo em Figura 2 e Figura 3, temos os formatos e o subconjunto de instruções do MIPS64, utilizando uma palavra de 64 bits, operandos localizados nos bancos de registradores, E/S, memória de dados ou de forma imediata:

Figura 2 – Formatos de instruções MIPS64

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa (1)

Figura 3 – Subconjunto de instruções do MIPS64

Tipo de instrução/opcode	Significado da instrução
Transferências de dados	*Move dados entre registradores e memória ou entre o inteiro e PF ou registradores especiais; somente o modo de endereço de memória é deslocamento de 16 bits + conteúdo de um GPR*
LB, LBU, SB	Load byte, load byte unsigned, store byte (de/para registradores inteiros)
LH, LHU, SH	Load half word, load half word unsigned, store half word (de/para registradores inteiros)
LW, LWU, SW	Load word, load word unsigned, store word (de/para registradores inteiros)
LD, SD	Load double word, store double word (de/para registradores inteiros)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copia de/para GPR para/de um registrador especial
MOV.S, MOV.D	Copia um registrador de PF (ponto flutuante) SP ou DP para outro registrador de PF
MFC1, MTC1	Copia 32 bits de/para registradores de PFpara/de registradores inteiros
Aritmética/lógica	*Operações sobre dados inteiros ou lógicos em GPRs; aritmética com sinal trap de overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (todos os imediatos são de 16 bits); com e sem sinal
DSUB, DSUBU	Subtração; com e sem sinal
DMUL, DMULU, DDIV, DDIVU, MADD	Multiplicação e divisão, com e sem sinal; multiply-add; todas as operações apanham e geram valores de 64 bits
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; carrega bits 32 a 47 do registrador com imediato, depois estende o sinal
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Deslocamentos: tanto imediato (DS__) quanto variável (DS__V); deslocamentos são shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate; com e sem sinal
Controle	*Desvios condicionais e saltos; relativo ao PC ou através do registrador*
BEQZ, BNEZ	Desvia se GPRs forem iguais/não iguais a zero; offset de 16 bits a partir de PC + 4
BEQ, BNE	Desvia se GPR forem iguais/não iguais; offset de 16 bits a partir de PC + 4
BC1T, BC1F	Testa bit de comparação no registrador de *status* de PF e desvia; offset de 16 bits a partir de PC + 4
MOVN, MOVZ	Copia GPR para outro GPR se terceiro GPR for negativo, zero
J, JR	Salto: offset de 26 bits a partir de PC + 4 (J) ou destino no registrador (JR)
JAL, JALR	Salto e link: salva PC + 4 em R31, destino é relativo ao PC (JAL) ou a um registrador (JALR)
TRAP	Transferência para sistema operacional em um endereço de vetor
ERET	Retorna ao código do usuário a partir de uma exceção; restaura modo do usuário
Ponto flutuante	*Operações de PF nos formatos DP e SP*
ADD.D, ADD.S, ADD.PS	Soma DP, números de SP e pares de números de SP
SUB.D, SUB.S, SUB.PS	Subtrai DP, números de SP e pares de números de SP
MUL.D, MUL.S, MUL.PS	Multiplica DP, ponto flutuante SP e pares de números de SP
MADD.D, MADD.S, MADD.PS	Multiplica-soma DP, números de SP e pares de números SP
DIV.D, DIV.S, DIV.PS	Divide DP, ponto flutuante de SP e pares de números de SP
CVT._._	Converte instruções: CVT.x.y converte do tipo x para tipo y, onde x e y são L (inteiro de 64 bits), W (inteiro de 32 bits), D (DP) ou S (SP). Ambos os operandos são FPRs.
C.__.D, C.__.S	Comparações de DP e SP: "__" = LT, GT, LE, GE, EQ, NE; marca bit no registrador de *status* de PF

Fonte: Arquitetura de Computadores: Uma Abordagem Quantitativa (1)

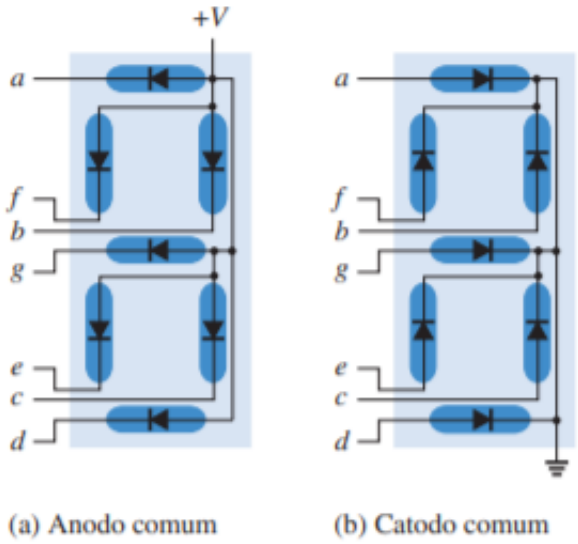
3.5 *Display* de 7 Segmentos

BCD, ou Codificação Binária Decimal, é uma forma de representar números decimais em binário de forma mais prática e evidente. Uma vez que segue de zero até nove para cada casa, a representação de demais casas em decimal é dada por dois números binários de 4 bits cada. Por exemplo: 9 em decimal, converte-se para, 1001 em BCD; porém 10 em decimal, converte-se para, 0001 0000 em BCD.

Uma vez que se sabe como funciona a conversão decimal para BCD, e o contrário,

temos que o *display* de sete segmentos não passa de uma forma mais ilustrativa de representação do BCD (muito utilizado em elevadores, por exemplo).

Figura 4 – *Display* de sete segmentos

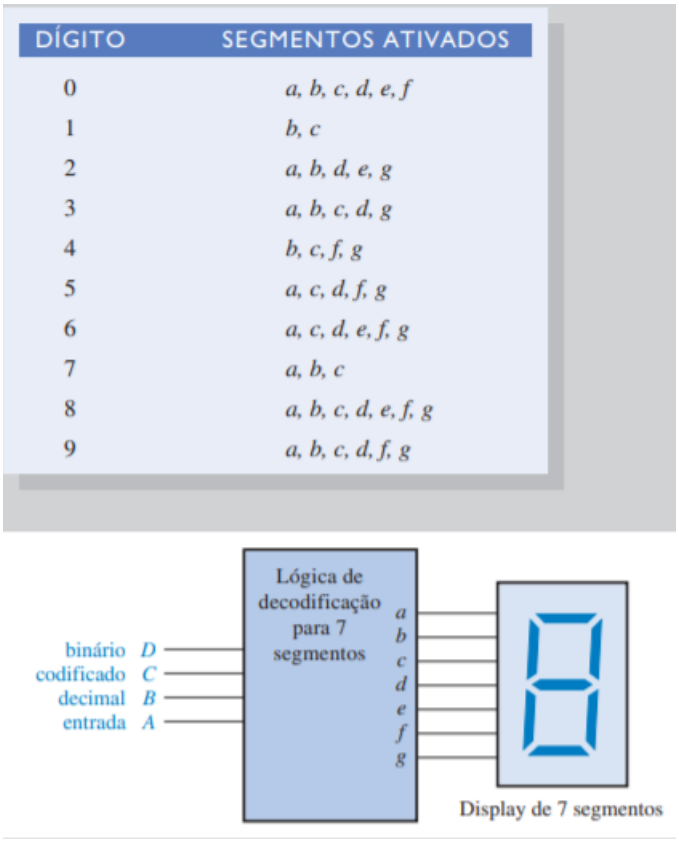


Fonte: Sistemas Digitais: Fundamentos e Aplicações (3)

Dessa forma, temos dois tipos de *displays* possíveis: Anodo Comum e o Catodo Comum. A principal diferença entre ambos é o fato de que o Anodo Comum é sua reação para os níveis de entrada, ou seja, entradas de nível alto para qualquer segmento do Anodo Comum o apagará, enquanto que, para o Catodo Comum o acenderá.

Logo, podemos fazer uma conversão de BCD para sete segmentos utilizando o *display*, através de um mapa de Karnaugh. Também conhecido como decodificador, teríamos (nota-se que tal exemplo utiliza-se de um *display* do tipo Catodo Comum):

Figura 5 – Conversor BCD - sete segmentos

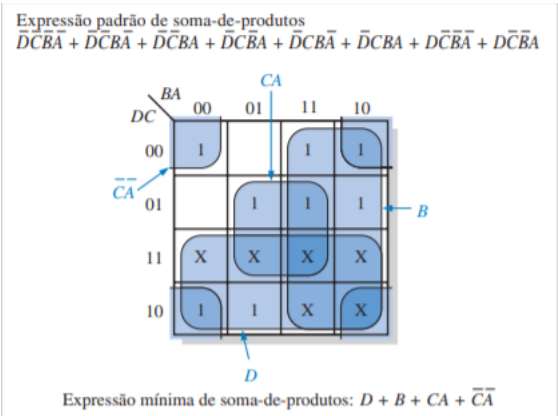


DÍGITO	SEGMENTOS ATIVADOS
0	*a, b, c, d, e, f*
1	*b, c*
2	*a, b, d, e, g*
3	*a, b, c, d, g*
4	*b, c, f, g*
5	*a, c, d, f, g*
6	*a, c, d, e, f, g*
7	*a, b, c*
8	*a, b, c, d, e, f, g*
9	*a, b, c, d, f, g*

Fonte: Sistemas Digitais: Fundamentos e Aplicações (3)

Considerando que: Saída = 1 significa ligado; Saída = 0 significa desligado; Saída = X significa "*don't care*". Seu mapa de Karnaugh, é algo do gênero:

Figura 6 – Minimização via mapa de Karnaugh



Fonte: Sistemas Digitais: Fundamentos e Aplicações (3)

Figura 7 – Tabela verdade sete segmentos

DÍGITO DECIMAL	ENTRADAS				SAÍDAS DOS SEGMENTOS						
	D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
10	1	0	1	0	X	X	X	X	X	X	X
11	1	0	1	1	X	X	X	X	X	X	X
12	1	1	0	0	X	X	X	X	X	X	X
13	1	1	0	1	X	X	X	X	X	X	X
14	1	1	1	0	X	X	X	X	X	X	X
15	1	1	1	1	X	X	X	X	X	X	X

Fonte: Sistemas Digitais: Fundamentos e Aplicações (3)

Exemplo de representação do número “0123456789” em dez *displays* de sete segmentos:

Figura 8 – “0123456789” em sete segmentos

Fonte: Sistemas Digitais: Fundamentos e Aplicações (3)

3.6 Quartus Prime e Altera D2-115

Todo conceito demonstrado deve ser executado em algum lugar e de alguma forma. A forma que foi decidida neste projeto é: utilizando o Quartus Prime e a placa reprogramável Altera De2-115. O Quartus Prime é um software da Intel, quando esta fez aquisição da Altera Quartus II da empresa Altera, esta ferramenta (também utilizada por nós neste projeto) permite o desenvolvimento de projetos e testes em placas reprogramáveis

do tipo Altera, sejam eles utilizando esquemáticos ou linguagens de programação de mais alto nível como o Verilog.

Já a placa Altera De2-115 é um FPGA (*Field-Programmable Gate Array*), ou seja, uma placa reprogramável (módulo físico de circuito digital) que nos permite estudar mais a fundo o conteúdo elucidado por meio de projetos e testes realizados diretamente nesta com auxilio do Quartus Prime.

3.7 Verilog

Verilog é uma linguagem de programação de alto nível padronizada pela IEEE-1364-2005. Mais especificamente, uma linguagem de descrição de hardware que é usada para modelar circuitos eletrônicos no geral, descrevendo tal circuito desejado e tendo a opção de simular este. Portanto esta foi a ferramenta escolhida para modelar o sistema computacional que desejamos projetar.

4 Desenvolvimento

4.1 Arquitetura Base

Procurando simplicidade e funcionalidade para desenvolver o sistema computacional, foi iniciada a busca por uma arquitetura com conjunto de instruções reduzidos (RISC), que pudesse ser implementada executando uma instrução a cada ciclo de *clock* (monociclo) e utilizasse o conceito da arquitetura de Harvard, ou seja, separando sua memória em duas partes: memória de instrução e memória de dados.

Dadas tais especificações, a arquitetura MIPS se apresentou a melhor opção para este projeto uma vez que contém formatos e conjunto de instruções simples e bem definidos, possível separação de memória a fim de seguir a arquitetura de Harvard e pode ser implementada de forma monocíclica, desta forma, foi escolhida como arquitetura base para todo o desenvolvimento do sistema computacional.

4.2 Conjunto de Instruções

A definição do conjunto de instruções foi cautelosamente dividido entre três etapas: palavra, formato e categoria. Sendo cada etapa, a sua maneira, responsável por trazer as instruções necessárias já existentes dentro do MIPS, e adaptá-las de forma que fossem necessárias para o projeto.

Na etapa um, a palavra de 32 bits, especificação que dá parte do nome da arquitetura específica “MIPS32”, foi escolhida para demarcar as instruções e, portanto, também limitar as linhas da memória de instruções a ser utilizada futuramente.

Na etapa dois, os formatos definidos, também trazidos do MIPS somados a palavra de 32 bits, foram classificados em três: tipo R (registrador), tipo I (imediato) e tipo J (jump), vide a Figura 9.

Figura 9 – Formatos de instrução

Tipo R	opcode [31-26]	rs [25-21]	rt [20-16]	rd [15-11]	shamt [10-6]	funct [5-0]
Tipo I	opcode [31-26]	rs [25-21]	rt [20-16]	imediato [15-0]		
Tipo J	opcode [31-26]	endereço [25-0]				

Fonte: Autor

Cada formato possui sua especificação, porém eles são segmentados de maneira similar a fim de ter 32 bits no total, para os tipos R e I: “opcode” de 6 bits, registradores

de 5 bits (no exemplo, “rs”, “rt” e “rd”), “shamt” de 5 bits, “funct” de 6 bits e um imediato de 16 bits; para o tipo J: “opcode” de 6 bits e um endereço de 26 bits.

Na etapa três, por opção de categorizar o conjunto de instruções a fim de facilitar na fase de escrita dos softwares planejados para este sistema computacional, foram planejadas as seguintes categorias de instruções: Aritméticas, referentes à todo e qualquer tipo de manipulação aritmética; Lógicas, referentes à todo e qualquer tipo de manipulação lógica; Condicionais, referentes à todo e qualquer tipo de manipulação condicional; Controle, referentes à todo e qualquer tipo de manipulação de movimentação e comunicação exterior.

Uma vez limitado, formatado e categorizado, o conjunto de instruções veio naturalmente da arquitetura base conforme as necessidades do projeto, considerando os seguintes modos de endereçamento:

1. Imediato: operando é uma constante dentro da instrução;

2. Registrador: operando está em um registrador;

3. Base-deslocamento: operando está em um local da memória cujo endereço é um registrador mais um valor constante dados pela instrução;

4. Relativo ao PC: instrução está no endereço que é a soma do PC à uma constante dada pela instrução;

5. Absoluto/Pseudo-direto: instrução está no endereço dado para o PC fazer o salto.

Figura 10 – Conjunto de instruções

Instrução	Tipo	Formato	Sintaxe	Operação
Adição	Aritmetico	R	add	R[d] ← R[s] + R[t]
Subtração	Aritmetico	R	sub	R[d] ← R[s] - R[t]
Multiplicação	Aritmetico	R	mult	R[d] ← R[s] * R[t]
Divisão	Aritmetico	R	div	R[d] ← R[s] / R[t]
Adição IMD	Aritmetico	I	addi	R[t] ← R[s] + IMD
Subtração IMD	Aritmetico	I	subi	R[t] ← R[s] + IMD
AND	Logico	R	and	R[d] ← R[s] & R[t]
OR	Logico	R	or	R[d] ← R[s] | R[t]
Negativa	Logico	R	not	R[d] ← ~R[s]
OR Exclusivo	Logico	R	xor	R[d] ← R[s] ^ R[t]
AND IMD	Logico	I	andi	R[t] ← R[s] & IMD
OR IMD	Logico	I	ori	R[t] ← R[s] | IMD
Desvio se igual	Condicional	I	beq	se (R[s] == R[t]), PC ← PC + 1 + IMD
Desvio se diferente	Condicional	I	bne	se (R[s] != R[t]), PC ← PC + 1 + IMD
Igual que	Condicional	R	set	R[d] ← (R[s] == R[t])
Menor que	Condicional	R	slt	R[d] ← (R[s] < R[t])
Desloca p/ esquerda	Condicional	R	sll	R[d] ← R[s] << IMD
Desloca p/ direita	Condicional	R	srl	R[d] ← R[s] >> IMD
Salto	Condicional	J	j	PC ← IMD
Salto REG	Condicional	R	jr	PC ← R[s]
Entrada	Controle	R	in	R[d] ← input
Saída	Controle	R	out	output ← R[s]
Armazenar	Controle	I	store	M[R[s] + IMD] ← R[t]
Carregar	Controle	I	load	R[t] ← M[R[s] + IMD]
Mover	Controle	R	move	R[d] ← R[s]
Mover IMD	Controle	I	movei	R[t] ← IMD

Fonte: Autor

4.3 Elaboração da Arquitetura Base

Resultando em: um PC que incrementa de maneira constante ou conforme os *jumps* e *branchs*; uma ULA, acrônimo para Unidade Lógica e Aritmética, cuja função é realizar toda e qualquer manipulação aritmética, lógica e de deslocamento utilizando o shamt do formato R (*shift amount*); um banco de registradores padronizado pela arquitetura base MIPS; uma memória de dados e uma memória de instruções.

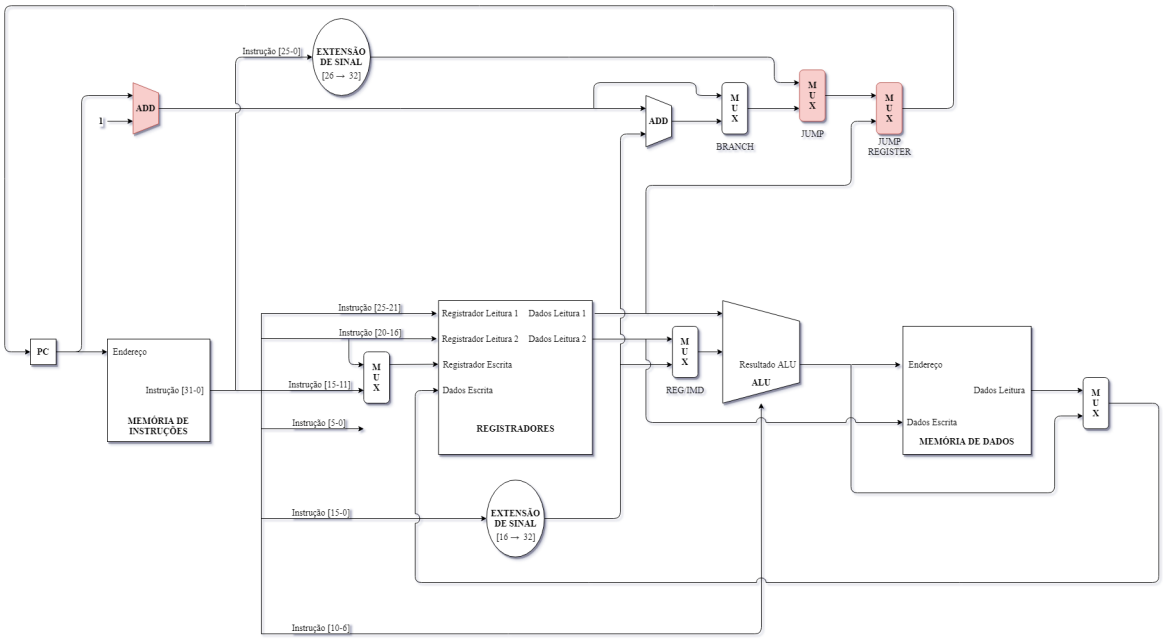
A elaboração da arquitetura base MIPS demonstrou um *datapath* muito semelhante ao original, apenas com algumas alterações que moldaram o projeto conforme havia necessidade, descrito abaixo e vide a Figura 11 (colorido em vermelho).

1. Incremento em 1 ao invés de 4 no PC uma vez que a memória de instrução tem 32 bits por linha ao invés de 4;

2. Multiplexador possibilitando *JUMP*, através de um endereço que apresenta incremento de 26 bits para 32 (formato de instrução J);

3. Multiplexador possibilitando *JUMP REGISTER*, através de um endereço dado por um registrador, localizado no banco de registradores.

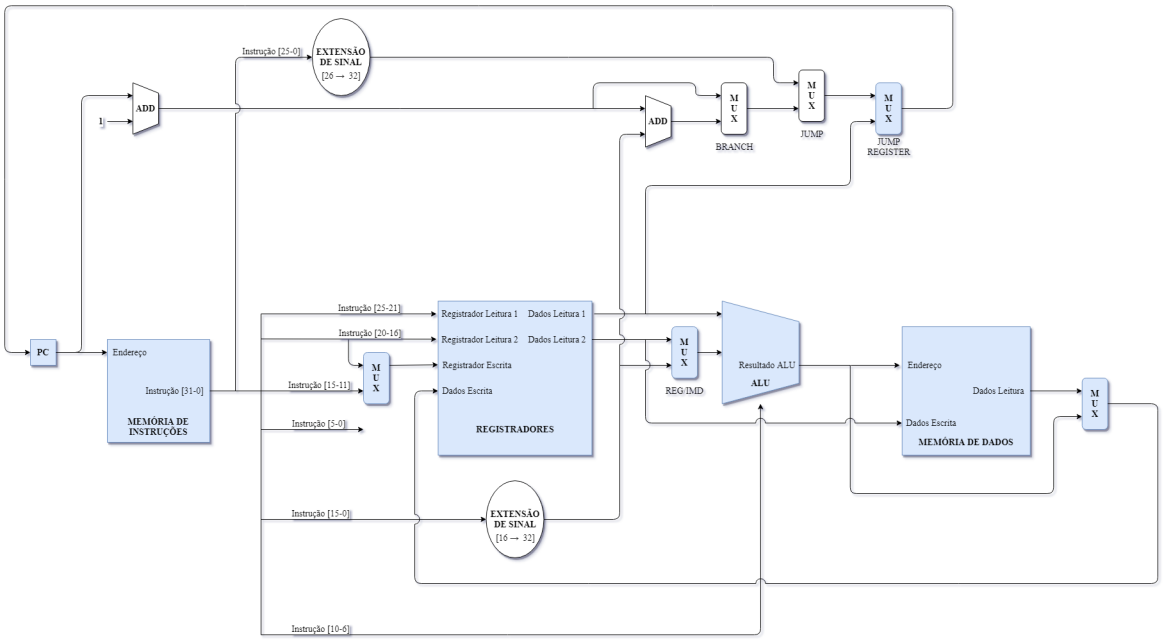
Figura 11 – *Datapath*



Fonte: Autor

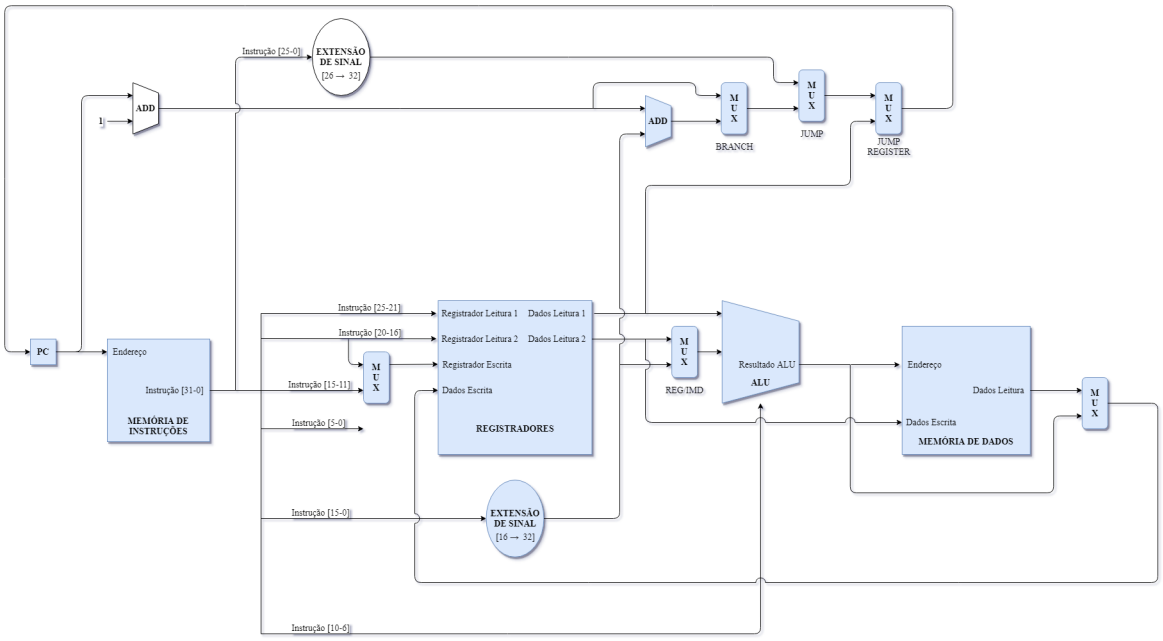
Enfim, com o *datapath* do sistema computacional elaborado, a consideração de uma unidade de controle conectada à cada multiplexador é indispensável (apesar de não representada para evitar poluição visual), uma vez que essa será a responsável pela decisão de fato do caminho dos dados para cada instrução aplicada através do ”opcode”, conforme pode ser verificado para formato possível de instrução existente:

Figura 12 – *Datapath* Tipo R



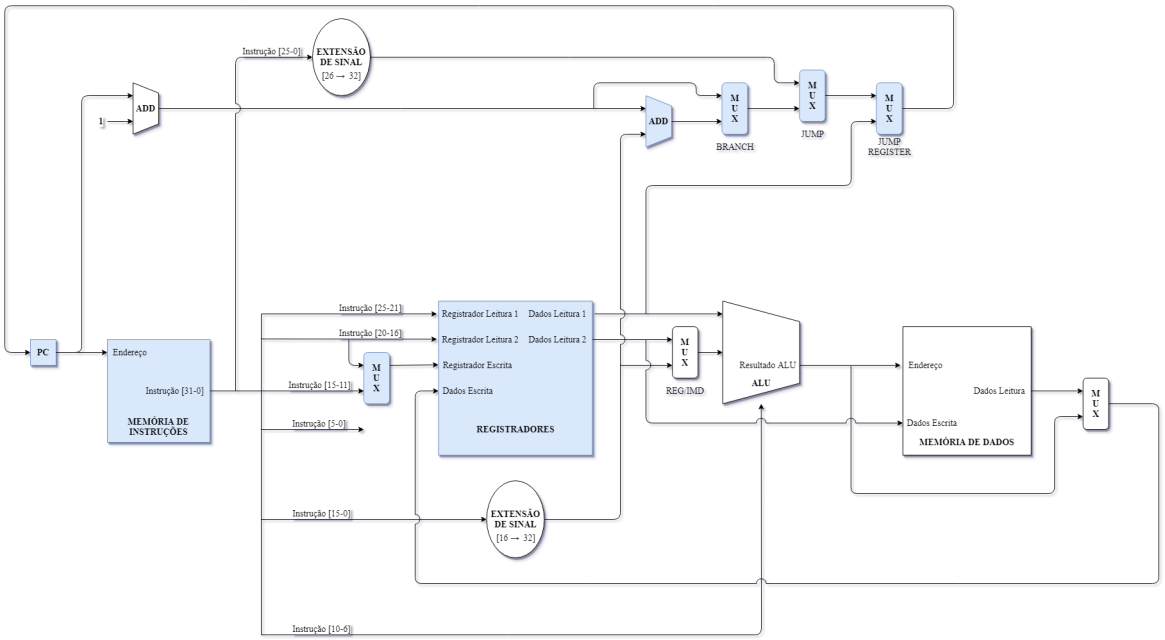
Fonte: Autor

Figura 13 – *Datapath* Tipo I



Fonte: Autor

Figura 14 – *Datapath* Tipo J



Fonte: Autor

Feita a análise individual de cada formato, foi constatado que: o PC, a memória de instruções e o banco de registradores são utilizados a cada instrução passada (portanto, a cada *clock*), uma vez que é necessário o uso destes para qualquer instrução apresentada.

Em compensação parte dos multiplexadores é fiel aos formatos I e J, uma vez que só valida ou não a instrução de *jump* requerida pela unidade de controle.

A ULA e a memória de dados por sua vez, segue o padrão dos formatos R e I, uma vez que atende à dados condicionais, lógicos e aritméticos.

4.4 Conjunto de Instruções atualizado

Já pensando na necessidade de organizar o conjunto de instruções previamente projetado para desenvolver corretamente a ULA e a UC, uma atualização foi feita em tal conjunto trazendo os ”opcodes”, ”functs” e valor da operação realizada na ULA que cada instrução necessitou, podendo ser observada na Figura 15.

Figura 15 – Conjunto de instruções atualizado

Instrução	Tipo	Formato	Sintaxe	Operação	Opcode	Funct	ULA
Entrada	Controle	R	in	R[d] ← input	x	000001	x
Saída	Controle	R	out	output ← R[s]	x	000010	x
Mover	Controle	R	move	R[d] ← R[s]	x	000011	0001
Adição	Aritmetico	R	add	R[d] ← R[s] + R[t]	x	000100	0100
Subtração	Aritmetico	R	sub	R[d] ← R[s] - R[t]	x	000101	0101
Multiplicação	Aritmetico	R	mult	R[d] ← R[s] * R[t]	x	000110	0110
Divisão	Aritmetico	R	div	R[d] ← R[s] / R[t]	x	000111	0111
AND	Logico	R	and	R[d] ← R[s] & R[t]	x	001000	1000
OR	Logico	R	or	R[d] ← R[s] | R[t]	x	001001	1001
Negativa	Logico	R	not	R[d] ← ~R[s]	x	001010	1010
OR Exclusivo	Logico	R	xor	R[d] ← R[s] ^ R[t]	x	001011	1011
Igual que	Condicional	R	set	R[d] ← (R[s] == R[t])	x	001100	1100
Menor que	Condicional	R	slt	R[d] ← (R[s] < R[t])	x	001101	1101
Desloca p/ esquerda	Condicional	R	sll	R[d] ← R[s] << IMD	x	001110	1110
Desloca p/ direita	Condicional	R	srl	R[d] ← R[s] >> IMD	x	001111	1111
Salto REG	Condicional	R	jr	PC ← R[s]	x	010000	x
Carregar	Controle	I	load	R[t] ← M[R[s] + IMD]	000001	x	0100
Armazenar	Controle	I	store	M[R[s] + IMD] ← R[t]	000010	x	0100
Mover IMD	Controle	I	movei	R[t] ← IMD	000011	x	0010
Adição IMD	Aritmetico	I	addi	R[t] ← R[s] + IMD	000100	x	0100
Subtração IMD	Aritmetico	I	subi	R[t] ← R[s] + IMD	000101	x	0101
AND IMD	Logico	I	andi	R[t] ← R[s] & IMD	001000	x	1000
OR IMD	Logico	I	ori	R[t] ← R[s] | IMD	001001	x	1001
Desvio se igual	Condicional	I	beq	se (R[s] == R[t]), PC ← PC + 1 + IMD	001100	x	1100
Desvio se diferente	Condicional	I	bne	se (R[s] != R[t]), PC ← PC + 1 + IMD	001101	x	0011
Salto	Condicional	J	j	PC ← IMD	010000	x	x

Fonte: Autor

4.5 Implementação do PC

Por se tratar de um registrador individual contendo 32 bits para memorização da posição atual da linha de código, posteriormente programado e utilizado para indicar ao processador quais instruções realizar, este foi projetado da seguinte forma:

- Uma entrada de 32 bits referente ao próximo valor de PC, oriundo do Multiplexador de PC, chamada **novo_PC**;
- Uma entrada referente ao ***clock*** do FPGA, cuja função primordial é alimentar o processador para que passe de um ciclo para o outro (mono-ciclicamente no neste caso);
- Uma saída de 32 bits que é um registrador chamado **PC**, que mantém o seu valor atual até a próxima descida do *clock*.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module PC (novo_PC, clock, PC);
2
3          input [31:0]novo_PC;     //Entrada do novo valor para PC
4          input clock;                      //Entrada do CLOCK
5          output reg [31:0]PC;     //Saida do PC
```

```
6
7           always @(negedge clock)
8                   begin
9                           PC <= novo_PC;          //PC = novo_PC
10                  end
11  endmodule
```

Portando, a cada descida de ***clock***, **PC** recebe o valor contido em **novo_PC**.

4.6 Implementação das Memórias

Por se tratar de memórias bem definidas conceitualmente, ambas foram implementadas a partir de modelos de RAM e ROM (respectivamente, acrônimos em inglês de Memória de Acesso Aleatório e Memória Somente de Leitura) advindas do próprio Quartus Prime, como pode ser visto a seguir.

4.6.1 Memória de Instruções

Na idealização da Memória de Instruções, onde foram armazenadas as instruções, foi utilizado um modelo já preexistente no Quartus Prime de "*Single Port* ROM", ou seja, uma memória que apenas lê um endereço recebido e passa adiante o dado existente neste:

- Uma entrada de 32 bits referente ao Endereço indicado pelo PC, chamada de **addr**;
- Uma entrada referente ao *clock* do FPGA (neste caso, por ser modelo do Quartus, chamada de **clk**);
- Uma saída de 32 bits referente ao Dado existente no endereço indicado, chamada de **q**;
- Um registrador matricial de 50 linhas e palavra de 32 bits chamado **rom** referente à memória ROM de fato, contendo as instruções.

Sendo sua implementação em Verilog da seguinte forma:

```
1   // Quartus Prime Verilog Template
2   // Single Port ROM
3
4   module Memoria_Instrucao
5   #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=32)
6   (
7           input [(ADDR_WIDTH-1):0] addr,
8           input clk,
9           output reg [(DATA_WIDTH-1):0] q
10  );
11
12          // Declare the ROM variable
13          reg [31:0] rom[49:0];
14
```

```
15          initial
16          begin
17                  //$readmemb("software.txt", rom);
18
19                  //TIPOS DE INSTRUCOES:
20                  //rom[0] <= 32'b000000_00000_00000_00000_00000_000000; //Tipo R
21                  //rom[0] <= 32'b000000_00000_00000_0000000000000000;     //Tipo I
22                  //rom[0] <= 32'b000000_00000000000000000000000000;       //Tipo J
23
24                  //FIBONACCI:
25
26                  rom[0] <= 32'b000011_00000_00101_0000000000000101;              //movei
27                  rom[1] <= 32'b000011_00000_00001_0000000000000001;              //movei
28                  rom[2] <= 32'b000011_00000_00010_0000000000000001;              //movei
29                  rom[3] <= 32'b000000_00001_00010_00011_00000_000100;    //add
30                  rom[4] <= 32'b000000_00010_00000_00001_00000_000011;    //move
31                  rom[5] <= 32'b000000_00011_00000_00010_00000_000011;    //move
32                  rom[6] <= 32'b000000_00001_00000_00000_00000_000010;    //out
33                  rom[7] <= 32'b000000_00010_00000_00000_00000_000010;    //out
34                  rom[8] <= 32'b000010_00000_00011_0000000000000010;              //store
35                  rom[9] <= 32'b000001_00000_00100_0000000000000010;              //load
36                  rom[10] <= 32'b000000_00100_00000_00000_00000_000010;   //out
37                  rom[11] <= 32'b000000_00101_00000_00000_00000_000010;   //out
38                  rom[12] <= 32'b001101_00100_00101_1111111111110110;     //bne
39                  rom[13] <= 32'b010000_00000000000000000000001101;               //jump
40          end
41
42          always @ (posedge clk)
43          begin
44                  q <= rom[addr];
45          end
46
47  endmodule
```

Onde **q** toma o valor do dado contido em **rom[addr]** a cada subida do *clock* **clk**. Sendo interessante citar que o bloco *Initial* define as instruções já "preexistentes" dentro desta memória, não tendo necessidade e muito menos sendo possível escrever nesta (uma vez que é ROM).

4.6.2 Memória de Dados

No caso da idealização da Memória de Dados, onde foi possível armazenar e carregar dados, foi utilizado um modelo já preexistente no Quartus Prime de "*Single Port* RAM", ou seja, uma memória volátil que tem a possibilidade de ler um endereço e informar o dado contido na memória neste endereço, ou de escrever um dado na memória no endereço informado:

- Uma entrada de 32 bits referente ao Dado indicado pelo Banco de Registradores, chamada de **data**;
- Uma entrada de 32 bits referente ao Endereço indicado pelo resultado da ULA, chamada de **addr**;

- Uma entrada referente ao controle vindo da UC, para saber se há ou não escrita naquela instrução dada, chamada de **we**;
- Uma entrada referente ao *clock* do FPGA (neste caso, por ser modelo do Quartus, chamada de **clk**);
- Uma saída de 32 bits referente ao Dado existente no endereço indicado, chamada de **q**;
- Um registrador matricial de 50 linhas e palavra de 32 bits chamado de **ram** referente à memória RAM de fato, contendo os dados já pre-carregados nesta ou escritos durante o processamento de instruções;
- Um registrador de 32 bits chamado de **addr_reg** referente à armazenação do ultimo endereço passado através de **addr**.

```
1  // Quartus Prime Verilog Template
2  // Single port RAM with single read/write address
3
4  module Memoria_Dados
5  #(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=32)
6  (
7          input [(DATA_WIDTH-1):0] data,
8          input [(ADDR_WIDTH-1):0] addr,
9          input we, clk,
10         output [(DATA_WIDTH-1):0] q
11 );
12
13         // Declare the RAM variable
14         reg [31:0] ram[31:0];
15
16         // Variable to hold the registered read address
17         reg [31:0] addr_reg;
18
19         always @ (posedge clk)
20         begin
21                 // Write
22                 if (we)
23                         ram[addr] <= data;
24
25                 addr_reg <= addr;
26         end
27
28         // Continuous assignment implies read returns NEW data.
29         // This is the natural behavior of the TriMatrix memory
30         // blocks in Single Port mode.
31         assign q = ram[addr_reg];
32
33 endmodule
```

Onde **q**, independente de **we**, sempre toma o valor de **ram[addr_reg]** (considerando que **addr_reg** contem o valor de **addr**) a cada subida do *clock* **clk**; e no caso de **we** estiver indicando que naquela instrução haver escrita na memória RAM, **ram[addr]** toma o valor de **data**.

4.7 Implementação dos Extensores

Sabendo que alguns valores provenientes de instruções enviadas da Memória de Instruções necessitariam de extensão de bits para alcançar exatos 32 e manter a simetria de dados, foram projetados dois extensores.

4.7.1 Extensor de 16 para 32 bits

O primeiro, desenvolvido para o número Imediato de 16 bits, dado por instruções tipo I, segue o seguinte padrão:

- Uma entrada de 16 bits referente ao Imediato vindo de uma instrução tipo I, chamada **Imediato_tam16**;
- Uma saída de 32 bits que é um registrador chamado **Imediato**, mantendo o valor dado pela instrução simetricamente.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Extensor_16 (imediato_tam16, imediato);
2
3          input [15:0]imediato_tam16; //Entrada do IMEDIATO com tamanho de 16 bits
4          output reg [31:0]imediato;  //Saida de IMEDIATO de 32 bits
5
6          always @(*)
7          begin
8                  if(imediato_tam16[15]) begin
9                          imediato = {16'b1111111111111111,imediato_tam16};
10                 end else begin
11                         imediato = {16'b0000000000000000,imediato_tam16};
12                 end
13         end
14 endmodule
```

Havendo uma análise do último bit de **Imediato_tam16** para saber se o número existente nele é positivo ou negativo, temos que a cada mudança deste, existem as seguintes opções dependendo de **Imediato_tam16**[**15**]:

- Caso 1: O número é negativo, **Imediato** torna-se uma concatenação de 16 bits de 0 e o **Imediato_tam16**;
- Caso 0: O número é positivo, **Imediato** torna-se uma concatenação de 16 bits de 1 e o **Imediato_tam16**.

4.7.2 Extensor de 26 para 32 bits

O segundo, desenvolvido para o Endereço de 26 bits, dado por instruções tipo J, segue o seguinte padrão:

- Uma entrada de 26 bits referente ao Endereço vindo de uma instrução tipo J, chamada **Endereco_tam26**;

- Uma saída de 32 bits que é um registrador chamado **Endereco**, mantendo o valor dado pela instrução simetricamente.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Extensor_26 (endereco_tam26, endereco);
2
3          input [25:0]endereco_tam26; //Entrada do ENDERECO de 26 bits
4          output reg [31:0]endereco;       //Saida de ENDERECO 32 bits
5
6          always @(*)
7          begin
8                  endereco = {6'b000000,endereco_tam26};
9          end
10 endmodule
```

Por se tratar de um numero natural não há necessidade de verificação de sinal, portanto a cada mudança da entrada, o registrador **Endereco** torna-se uma concatenação de 6 bits de 0 e **Endereco_tam26**.

4.8 Implementação dos Multiplexadores

Com a transição de vários dados simultaneamente, foi necessário o desenvolvimento de multiplexadores que, com controladores oriundos da UC, indicam qual dado se utilizar dependendo da instrução dada, totalizando em quatro multiplexadores primordiais.

4.8.1 Multiplexador do PC

Com o intuito de decidir qual será a próxima posição do PC, foi planejado e desenvolvido o multiplexador deste. Possibilitando quatro opções no total, o multiplexador contém os seguintes elementos:

- Uma entrada de 32 bits referente ao valor atual do PC, chamada **PC**;

- Uma entrada de 32 bits referente ao Imediato vindo de uma instrução tipo I, chamada de **Imediato**;

- Uma entrada de 32 bits referente ao Endereço vindo de uma instrução tipo J, chamada de **Endereco**;

- Uma entrada de 32 bits referente ao Dado vindo do Banco de Registradores, chamada de **Dado_1**;

- Uma entrada referente ao controle vindo da UC, para saber qual será o valor de **Novo_PC**, chamada de **Mux_PC**;
- Uma saída de 32 bits que é um registrador chamado **Novo_PC**, que indica o valor da próxima posição de PC (dependendo da indicação dada por **Mux_PC**).

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Multiplex_PC (PC, imediato, endereco, dado_1, mux_PC, novo_PC);
2
3          input [31:0]PC;                          //Entrada do PC
4          input [31:0]imediato;           //Entrada do IMEDIATO
5          input [31:0]endereco;           //Entrada do ENDERECO
6          input [31:0]dado_1;                      //Entrada do DADO 1
7          input   [1:0]mux_PC;                     //Entrada do CONTROLADOR
8          output reg [31:0]novo_PC;       //Saida do novo valor para PC
9
10         always @(*)
11         begin
12                 case(mux_PC)
13                         0:
14                                 novo_PC = PC + 1;
                                                                  //CASO 0: PC = PC + 1
15                         1:
16                                 novo_PC = PC + 1 + $signed(imediato);          //CASO 1:
                                       PC = PC + 1 + IMEDIATO
17                         2:
18                                 novo_PC = endereco;
                                                        //CASO 2: PC = ENDERECO
19                         3:
20                                 novo_PC = dado_1;
                                                                  //CASO 3: PC = R[s]
21                 endcase
22         end
23 endmodule
```

Onde, sensível à qualquer entrada, o novo valor de PC depende da instrução que está sendo processada e é indicada pelo **Mux_PC** em quatro casos dependendo do valor deste:

- Caso 0: O próximo valor de PC acresce em apenas uma linha, ou seja, **Novo_PC** toma o valor de PC mais 1;
- Caso 1: O próximo valor de PC se baseia num *branch*, ou seja, dependendo de uma comparação realizada na ULA, **Mux_PC** toma o valor de PC mais 1 mais o Imediato dado pela instrução tipo I;
- Caso 2: O próximo valor de PC se baseia num *jump*, ou seja, **Mux_PC** toma o valor do **Endereco** dado pela instrução tipo J;
- Caso 3: O próximo valor de PC se baseia num *jumpR*, ou seja, **Mux_PC** toma o valor de **Dado_1** dado pela instrução tipo J (nesse caso um pulo para um dado contido num registrador).

4.8.2 Multiplexador do Registrador de Escrita

Com o intuito de decidir qual Endereço dar para realizar uma possível escrita no Banco de Registradores, foi planejado e desenvolvido um multiplexador para tal escolha. Possibilitando três opções no total, o multiplexador contém os seguintes elementos;

- Uma entrada de 5 bits referente ao Endereço de "R[t]" vindo de instruções tipo I, chamada de **Endereco_2**;
- Uma entrada de 5 bits referente ao Endereço de "R[d]" vindo de instruções tipo R, chamada de **Endereco_3**;
- Uma entrada referente ao controle vindo da UC, para saber qual Endereço deverá ser passado para o Banco de Registradores, chamada de **Mux_REGISTRADOR**;
- Uma saída que é um registrador de 5 bits chamado **Endereco_Escrita**, que indica qual Endereço deverá ser passado para a possível escrita no Banco de Registradores.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Multiplex_OperandoREG (endereco_2, endereco_3, mux_REGISTRADOR, endereco_escrita);
2
3          input [4:0]endereco_2;                                  //Entrada do ENDERECO por
               R[t]
4          input [4:0]endereco_3;                                  //Entrada do ENDERECO por
               R[d]
5          input [1:0]mux_REGISTRADOR;                     //Entrada do CONTROLADOR
6          output reg [4:0]endereco_escrita;       //Saida do ENDERECO de escrita
7
8          always @(*)
9          begin
10                 case(mux_REGISTRADOR)
11                         0:
12                                 endereco_escrita = 0;
13                         1:
14                                 endereco_escrita = endereco_2;
15                         2:
16                                 endereco_escrita = endereco_3;
17                 endcase
18         end
19 endmodule
```

Onde, sensível a qualquer entrada, dependendo do valor de **Mux_REGISTRADOR**, há três possibilidades:

- Caso 0: O **Endereco_Escrita** toma o valor de 0, indicando que não há endereço para ser escrito, apresentando o endereço do registrador fixo com valor zero;
- Caso 1: O **Endereco_Escrita** toma o valor de **Endereco_2**, indicando que a instrução é tipo I e entregando o endereço do registrador R[t] escolhido;

- Caso 2: O **Endereco_Escrita** toma o valor de **Endereco_3**, indicando que a instrução é tipo R e entregando o endereço do registrador R[d] escolhido.

4.8.3 Multiplexador do Dado para a ULA

Com o intuito de decidir qual será o Dado entregue a ULA para possivelmente realizar seus cálculos (realizando na grande maioria das instruções), foi planejado e desenvolvido um multiplexador para tal escolha. Possibilitando duas opções no total, o multiplexador contém os seguintes elementos;

- Uma entrada de 32 bits referente ao segundo Dado vindo do Banco de Registradores, chamada de **Dado_2**;
- Uma entrada de 32 bits referente ao Imediato vindo de uma instrução tipo I, chamada de **Imediato**;
- Uma entrada referente ao controle vindo da UC, para saber qual Dado deverá ser passado para a ULA, chamada de **Mux_ULA**;
- Uma saída que é um registrador de 32 bits chamado **Dado_ULA**, que indica qual Dado deverá ser passado para a possível realização de cálculo na ULA.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Multiplex_OperandoULA (dado_2, imediato, mux_ULA, dado_ULA);
2
3          input [31:0]dado_2;                              //Entrada do DADO 2
4          input [31:0]imediato;                    //Entrada do IMEDIATO
5          input mux_ULA;                                   //Entrada do CONTROLADOR
6          output reg [31:0]dado_ULA;               //Saida do DADO ULA
7
8          always @(*)
9          begin
10                 case(mux_ULA)
11                         0:
12                                 dado_ULA = dado_2;
13                         1:
14                                 dado_ULA = imediato;
15                 endcase
16         end
17 endmodule
```

Onde, sensível a qualquer entrada, dependendo do valor de **Mux_ULA**, há duas possibilidades:

- Caso 0: O **Dado_ULA** toma o valor de **Dado_2**, indicando que a instrução é do tipo R e os Dados utilizados na ULA serão ambos do Banco de Registradores;

- Caso 1: O **Dado_ULA** toma o valor de **Dado_2**, indicando que a instrução é do tipo I e os Dados utilizados na ULA serão um do Banco de Registradores e outo um Imediato.

4.8.4 Multiplexador do Dado de Escrita

Com o intuito de decidir qual será o Dado entregue para o Banco de Registradores para realizar uma possível escrita neste (nota-se a relação direta com o Multiplexador do Registrador de Escrita, um define ”onde” e outro define ”o que”), foi planejado e desenvolvido um multiplexador para tal escolha. Possibilitando três opções no total, o multiplexador contém os seguintes elementos;

- Uma entrada de 32 bits referente ao resultado dado pela ULA, chamada de **Resultado_ULA**;
- Uma entrada de 32 bits referente à um dado vindo da Memória de Dados, chamada de **Memoria_Dados**;
- Uma entrada de 32 bits referente à um dado vindo do *input* do usuário, chamada de **Dado_Escrita**;
- Uma entrada referente ao controle vindo da UC, para saber qual Dado deverá ser passado para o Banco de Registradores, chamada de **Mux_ESCRITA**;
- Uma saída que é um registrador de 32 bits chamado **Dado_Escrita**, que indica qual Dado deverá ser passado para a possível escrita no Banco de Registradores.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Multiplex_OperandoESCRITA (resultado_ULA, memoria_dados, dado_entrada, mux_ESCRITA
       , dado_escrita);
2
3          input [31:0]resultado_ULA;                      //Entrada do DADO do resultado da
               ULA
4          input [31:0]memoria_dados;                      //Entrada do DADO da memoria de
               dados
5          input [31:0]dado_entrada;                       //Entrada do DADO do input
6          input [1:0]mux_ESCRITA;                         //Entrada do CONTROLADOR
7          output reg [31:0]dado_escrita;  //Saida do DADO que sera escrito
8
9          always @(*)
10                 begin
11                         case(mux_ESCRITA)
12                                 0:
13                                         dado_escrita = resultado_ULA;
14                                 1:
15                                         dado_escrita = memoria_dados;
16                                 2:
17                                         dado_escrita = dado_entrada;
18                         endcase
```

```
19                  end
20  endmodule
```

Onde, sensível a qualquer entrada, dependendo do valor de **Mux_ESCRITA**, há três possibilidades:

- Caso 0: O **Dado_Escrita** toma o valor de **Resultado_ULA**, indicando que o Dado a ser possivelmente escrito deverá ser o resultado da ULA;
- Caso 1: O **Dado_Escrita** toma o valor de **Memoria_Dados**, indicando que o Dado a ser possivelmente escrito deverá ser aquele vindo da Memoria de Dados;
- Caso 3: O **Dado_Escrita** toma o valor de **Dado_Entrada**, indicando que o Dado a ser possivelmente escrito deverá ser o *input* dado pelo usuário.

4.9 Implementação do Banco de Registradores

Como proposto inicialmente no projeto, houve a necessidade de ter um espaço utilizado a todo momento para transicionar os dados e, diferente da Memória de Dados, salvar temporariamente estes (salvo os valores fixos usados para comparações e afins). Portando, chegando ao ponto de projetar e desenvolver um Banco de Registradores com os seguintes elementos:

- Uma entrada de 5 bits referente ao Endereço 1 dado por uma instrução do tipo R ou I, ou seja ”s” de R[s], chamada de **Endereco_1**;
- Uma entrada de 5 bits referente ao Endereço 2 dado por uma instrução do tipo R ou I, ou seja ”t” de R[t], chamada de **Endereco_1**;
- Uma entrada de 5 bits referente ao Endereço de Escrita dado pelo Multiplexador do Registrador de Escrita (podendo ser ”t” ou ”d”), chamada de **Endereco_Escrita**;
- Uma entrada de 32 bits referente ao Dado de Escrita dado pelo Multiplexador do Dado de Escrita, chamada de **Dado_Escrita**;
- Uma entrada referente ao e chamada de ***clock*** do FPGA;
- Uma entrada referente ao controle vindo da UC, para saber se deverá ou não escrever o **Dado_Escrita** no **Endereco_Escrita**, chamada de **Controle_REGISTRADOR**;
- Uma saída de 32 bits que passa adiante o Dado existente em **Registrador**[**Endereco_1**], chamada de **Dado_1**;
- Uma saída de 32 bits que passa adiante o Dado existente em **Registrador**[**Endereco_2**], chamada de **Dado_2**;

- Um registrador matricial de 32 Registradores com tamanho de 32 bits, sendo este o Banco de Registradores chamado de **Registrador**.

Sendo sua implementação em Verilog da seguinte forma:

```
1  module Banco_Registradores (endereco_1, endereco_2, endereco_escrita, dado_escrita, clock
       , controle_REGISTRADOR, dado_1, dado_2);
2
3          input [4:0]endereco_1;                  //Entrada do ENDERECO do registrador 1 (R
               [s])
4          input [4:0]endereco_2;                  //Entrada do ENDERECO do registrador 2 (R
               [t])
5          input [4:0]endereco_escrita;    //Entrada do ENDERECO do registrador (R[t]/R[d])
6          input [31:0]dado_escrita;               //Entrada do DADO a ser escrito
7          input clock;                                            //Entrada do CLOCK
8          input controle_REGISTRADOR;     //Entrada do CONTROLADOR
9          output signed [31:0]dado_1;     //Saida do DADO 1 a ser utilizado
10         output signed [31:0]dado_2;     //Saida do DADO 2 a ser utilizado
11
12         reg [31:0]registrador[31:0];    //Banco de 32x32 Registradores (REGISTRADOR)
13
14         initial registrador[32'd0] = 0;
15
16         always @(posedge clock)
17         begin
18                 if(controle_REGISTRADOR) registrador[endereco_escrita] = dado_escrita;
19         end
20
21         assign dado_1 = registrador[endereco_1];
22         assign dado_2 = registrador[endereco_2];
23 endmodule
```

Sabendo que a escrita no Banco de Registradores é sensível à mudança apenas na subida ***clock***, o **Dado_Escrita** é inserido no **Registrador[Endereco_Escrita]** apenas quando o **Controle_REGISTRADOR** apresenta o valor 1 em tal mudança, porém os Dados passados adiante estão sempre atualizados de acordo com os endereços dados. É interessante a ressalva de que (como pode ser visto no bloco *Initial*) o **Registrador[0]** começa e se mantem com o valro 0 dentro de si.

4.10 Implementação da ULA

Por se tratar de uma implementação muito extensa, esta foi demarcada como um apêndice deste relatório e se encontra em Apêndice A. A ULA foi projetada e implementada para realizar todas as instruções aritméticas, lógicas e algumas de controle e condicionais (mais especificamente todas as instruções que contem algum "opcode" de ULA descrito na Figura 15). Contendo os seguintes elementos:

- Uma entrada de 32 bits referente ao Dado 1 vindo do Banco de registradores, chamada de **Dado_1**;

- Uma entrada de 32 bits referente ao Dado vindo do multiplexador de dados para ULA, chamada de **Dado_ULA**;
- Uma entrada de 5 bits referente ao *shamt* vindo de uma instrução do tipo R, chamada **Shamt**;
- Uma entrada referente ao controle vindo da UC, para saber qual a função deverá ser realizada envolvendo os dados citados, chamada de **Controle_ULA**;
- Uma saída que é um registrador de 32 bits chamado de **Resultado_ULA**, que indica o resultado da função selecionada utilizando os dados passados;
- Uma saída que é um registrador chamado **Zero** que é conhecido como *flag zero*, indicando normalmente resultado de comparações e muito utilizado em *branchs*.

Como foi apresentado no conjunto de instruções, dependendo do **Controle_ULA**, a ULA realizará uma operação utilizando ambas as entradas (ou apenas uma) e colocará tal resultado em **Resultado_ULA**, passando adiante tal resultado. Um ponto interessante é o **Shamt** ser utilizado apenas em operações de deslocamento de bits; assim como vale ressaltar que o bloco *Always* é sensível à qualquer entrada, realizando a operação desejada a qualquer mudança de entrada apresentada.

4.11 Implementação da UC

Por se tratar de uma implementação muito extensa, esta foi demarcada como um apêndice deste relatório e se encontra em Apêndice B. Sendo o cérebro e tradutor de instruções submetidas pela Memória de Instruções pre-escrita pelo programador, a Unidade de Controle (UC) foi implementada com os seguintes elementos:

- Uma entrada de 32 bits referente à uma instrução submetida pela Memória de Instruções, chamada de **Instrucao**;
- Uma entrada referente à *flag* (e chamada de) **Zero**;
- Uma saída que é um registrador chamado de **Controle_Entrada**, indica se há *input* do usuário;
- Uma saída que é um registrador chamado de **Controle_Saida**, indica se há *output* para o usuário;
- Uma saída que é um registrador chamado de **Controle_ULA** contendo 5 bits, indica qual é a operação a ser realizada na ULA;

- Uma saída que é um registrador chamado de **Controle_MemoriaDADOS**, indica se há ou não escrita na Memoria de Dados;
- Uma saída que é um registrador chamado de **Controle_REGISTRADOR**, indica se há ou não escrita no Banco de Registradores;
- Uma saída de um registrador chamado de **Mux_PC**, usado para controlar qual escolha o Multiplexador do PC deve tomar;
- Uma saída de um registrador chamado de **Mux_ULA**, usado para controlar qual escolha o Multiplexador de Dado para a ULA deve tomar;
- Uma saída de um registrador chamado de **Mux_REGISTRADOR**, usado para controlar qual escolha o Multiplexador de Endereço do Registrador de scrita deve tomar;
- Uma saída de um registrador chamado de **Mux_ESCRITA**, usado para controlar qual escolha o Multiplexador de Dado de escrita deve tomar;

Sabendo que os blocos *Always* são sensíveis às entradas **Instrucao** e **Zero**, a implementação da UC gira em torno destes dados. Traduzindo **Instrucao** para o "opcode" e, quando necessário, trazendo "funct", através dos controladores indica qual instrução está sendo executada e quais dados devem transitar pelo processador e como.

4.12 Implementação da interface de E/S

Como não foi possível durante o transitar do projeto dispor de um kit FPGA em mãos, as interfaces foram simuladas em Verilog para entrar com valores pela "entrada" e apresentar valores pela "saída" digitalmente.

4.12.1 Interface de Entrada

Tendo como objetivo dar valores diretos para serem escritos no Banco de Registradores (instrução *in*) na simulação, a Interface de Entrada foi projetada da seguinte maneira:

- Uma entrada de 32 bits referente ao dado de *input* vindo do usuário, chamada de **Entrada**;
- Uma entrada referente ao controle vindo da UC, para saber se deverá ser usado **Entrada** ou não, chamada de **Controle_Entrada**;
- Uma saída de 32 bits que é um registrador chamado de **Nova_Entrada**, passa o valor de **Entrada** caso deva (indicado por **Controle_Entrada**).

Sendo sua implementação em Verilog da seguinte forma:

```
module Entrada (entrada, controle_entrada, nova_entrada);

        input [31:0]entrada;                                    //Entrada do dado para
            input
        input controle_entrada;                         //Entrada do CONTROLE de input
        output reg [31:0]nova_entrada;  //Saida do input

        always @(*)
                begin
                        if(controle_entrada) nova_entrada = entrada;
                        else nova_entrada = 0;
                end

endmodule
```

Onde, sensível a mudança de qualquer entrada, caso **Controle_Entrada** indique com o valor 1, **Nova_Entrada** recebe o valor de **Entrada**, caso contrário, recebe o valor zero.

4.12.2 Interface de Saída

Tendo como objetivo receber valores do Banco de Registradores para serem apresentados (instrução *out*) na simulação, a Interface de Saída foi projetada da seguinte maneira:

- Uma entrada de 32 bits referente ao dado vindo do Banco de Registradores, chamada de **Dado_1**;
- Uma entrada referente ao controle vindo da UC, para saber se deverá ser mostrado **Dado_1** ou não, chamada de **Controle_Saida**
- Uma saída de 32 bits que é um registrador chamado de **Saida**, apresentando o valor dado por **Dado_1** na simulação como um *output*.

Sendo sua implementação em Verilog da seguinte forma:

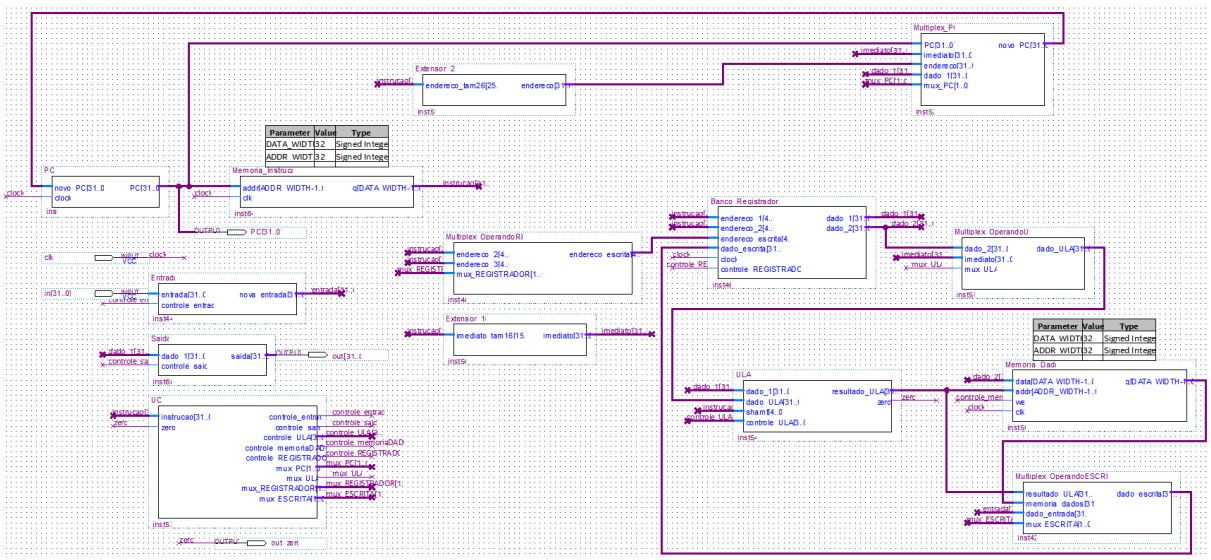
```
module Saida (dado_1, controle_saida, saida);

        input [31:0]dado_1;             //Entrada do dado para output
        input controle_saida;   //Entrada do CONTROLE de output
        output reg [31:0]saida; //Saida do output


        always @(*)
                begin
                        if(controle_saida) saida = dado_1;
                        else saida = 0;
                end

endmodule
```

Onde, sensível a mudança de qualquer entrada, caso **Controle_Saida** indique com o valor 1, **Saida** recebe o valor de **Dado_1**, caso contrário, recebe o valor zero.

4.13 Concatenação dos módulos implementados

Uma vez que o projeto tinha todos os módulos planejados implementados de fato em lógica programável Verilog, a concatenação destes foi realizada num arquivo ".bdf" pela facilidade proposta em tal tipo de esquematização e pela forma mais didática e visual de como o processador se monta pelas entradas e saídas descritas durante as implementações individuais. Atribuindo conforme o *datapath* primordialmente planejado incitava, juntamente com a atualização da implementação de uma UC, o processador inteiro pode ser vista em Figura 16.

Figura 16 – Concatenação do processador esquematizado

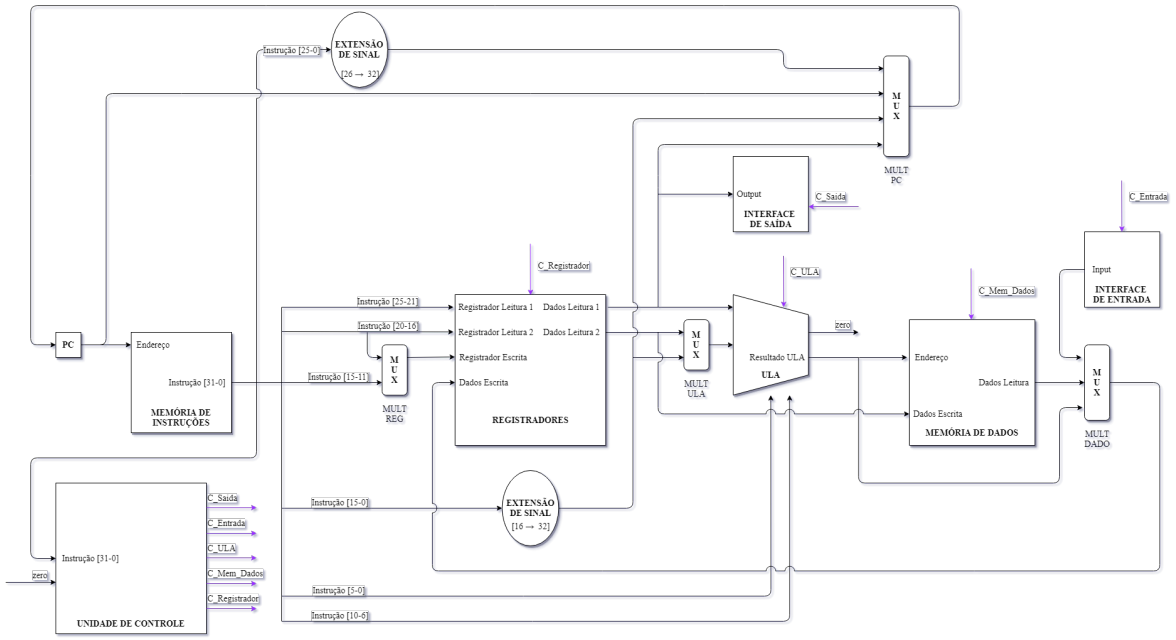


Fonte: Autor

4.14 Nova esquematização do *Datapath*

Apenas realizando pequenas alterações que foram necessárias durante a implementação, sendo a maior delas a introdução de uma UC e as interfaces E/S, um novo *datapath* foi elaborado para coincidir com a concatenação do processador e demonstrar de forma mais minimalista e bem organizada tal processador, que pode ser vista em Figura 17.

Figura 17 – *Datapath* atualizado



Fonte: Autor

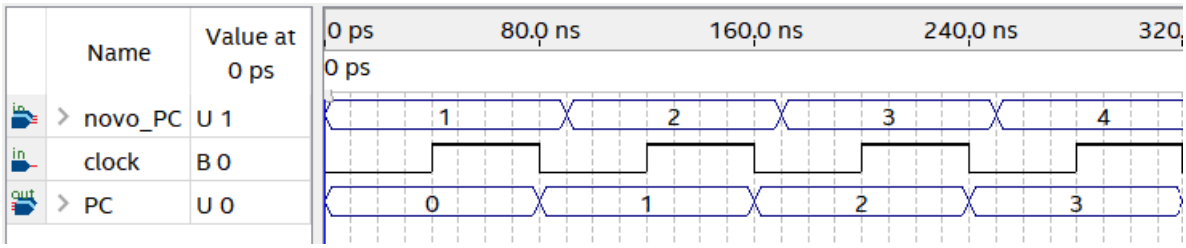
É importante a ressalva de que todos os multiplexadores tem também sinais de controle oriundos da UC, porém não foram explicitados para melhor visibilidade dos principais elementos do *datapath*.

5 Resultados Obtidos e Discussões

Na intenção de realizar testes no sistema computacional projetado individualmente, ou seja de forma modular, e concatenado, utilizamos o Quartus Prime e suas simulações em formas de onda. Dessa forma pudemos gerar os seguintes resultados.

5.1 Simulação do PC

Figura 18 – Simulação do PC

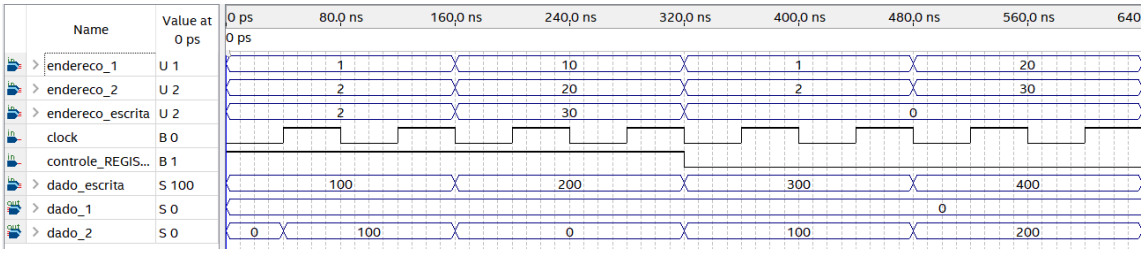


Fonte: Autor

Realizada a simulação, que pode ser vista na Figura 18, foi utilizado um **Novo_PC**, inserido manualmente, que indica para o **PC** qual valor ele deve ser na descida do ***clock*** conforme vista na implementação. Baseando e comparando com a proposta feita na implementação com o resultado da simulação, este módulo, individualmente, está de acordo com o comportamento apresentado, confirmando o fato de ser um PC funcional.

5.2 Simulação do Banco de Registradores

Figura 19 – Simulação do Banco de Registradores



Fonte: Autor

Realizada a simulação, que pode ser vista na Figura 19, foram utilizados, a cada dois ***clocks***, valores variados inseridos manualmente nos três endereços de entrada possíveis

(**Endereco_1**, **Endereco_2** e **Endereco_Escrita**), juntamente de um sinal positivo para **Controle_REGISTRO** na primeira metade da simulação (permitindo a escrita no **Endereco_Escrita** no valor de **Dado_Escrita**, também inserido manualmente) nos seguintes cenários:

- Cenário 1:

  Endereco_1 recebe 1;

  Endereco_2 recebe 2;

  Endereco_Escrita recebe 2;

  Controle_REGISTRO recebe 1, autorizando a escrita na subida do ***clock***;

  Dado_Escrita recebe 100;

  Dado_1 apresenta 0;

  Dado_2 apresenta 0 até a primeira subida, e 100 após esta.

- Cenário 2:

  Endereco_1 recebe 10;

  Endereco_2 recebe 20;

  Endereco_Escrita recebe 30;

  Controle_REGISTRO recebe 1, autorizando a escrita na subida do ***clock***;

  Dado_Escrita recebe 200;

  Dado_1 apresenta 0;

  Dado_2 apresenta 0.

- Cenário 3:

  Endereco_1 recebe 1;

  Endereco_2 recebe 2;

  Endereco_Escrita recebe 0;

  Controle_REGISTRO recebe 0, negando a escrita na subida do ***clock***;

  Dado_Escrita recebe 300;

  Dado_1 apresenta 0;

  Dado_2 apresenta 100, provando que houve a escrita e esta se mantém desde o "Cenário 1".

- Cenário 4:

  Endereco_1 recebe 20;

Endereco_2 recebe 30;

Endereco_Escrita recebe 0;

Controle_REGISTRO recebe 0, negando a escrita na subida do ***clock***;

Dado_Escrita recebe 400;

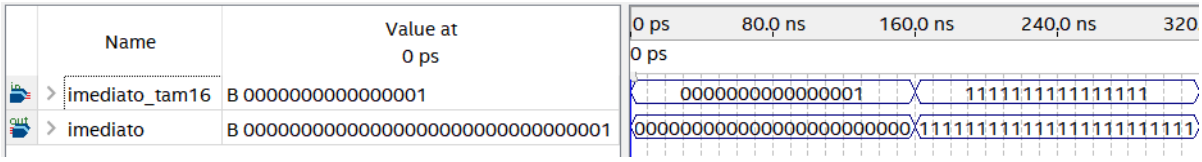
Dado_1 apresenta 0;

Dado_2 apresenta 200, provando que houve escrita e esta se mantém desde o "Cenário 2".

Comprovada a proposição, pela simulação dos cenários dados, feita durante o desenvolvimento da possibilidade de ler os dados existentes dentro dos endereços dados e escrever dados inseridos nos registradores desejados, portanto o módulo em questão apresenta individualmente comportamento esperado.

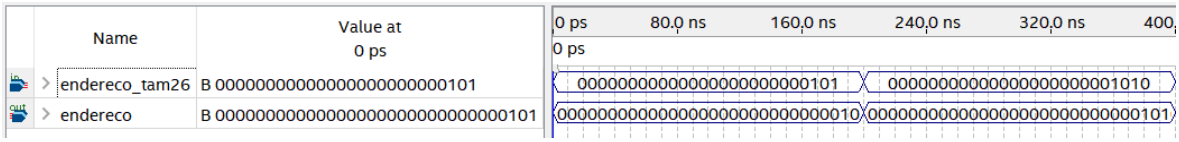
5.3 Simulação dos Extensores

Figura 20 – Simulação do Extensor de 16 bits



Fonte: Autor

Figura 21 – Simulação do Extensor de 26 bits



Fonte: Autor

Conforme apresentado na Figura 20 e na Figura 21, a simulação dos extensores de bits para um Imediato e para um Endereço tiveram valores de entrada inseridos manualmente da seguinte forma:

- Extensor de 16 bits: Neste caso, os valores inseridos em **Imediato_tam16** foram um positivo e um negativo por complemento de 2, sendo seu resultado, o mesmo numero inserido apenas estendido até 32 bits;

- Extensor de 26 bits: Neste caso, os valores inseridos em **Imediato_tam26** foram ambos positivos porém diferentes (uma vez que para o Endereço de PC, não existe valor negativo), sendo seu resultado, o mesmo numero inserido apenas estendido até 32 bits.

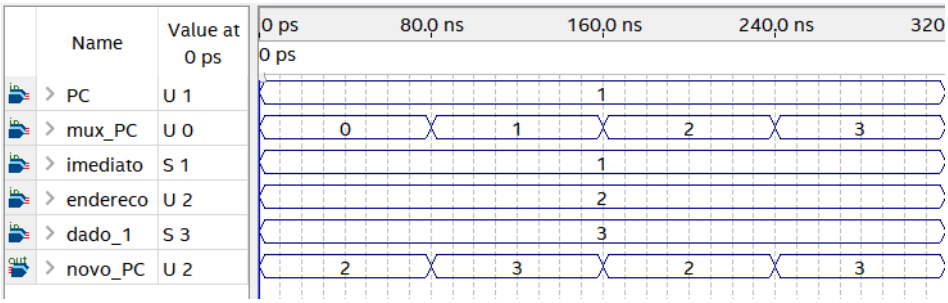
Conforme a simulação apresentou em ambos os casos, a proposta feita durante o desenvolvimento de apenas estender os valores dados na intenção de manter a simetria de 32 bits, sejam esses positivos ou negativos (para o caso do Imediato) foi mantida, portanto o módulo em questão apresenta individualmente comportamento esperado.

5.4 Simulação dos Multiplexadores

Realizando simulações individualmente em cada multiplexador implementado, foram obtidas as seguintes *waveforms*.

5.4.1 Multiplexador do PC

Figura 22 – Simulação do Multiplexador do PC



Fonte: Autor

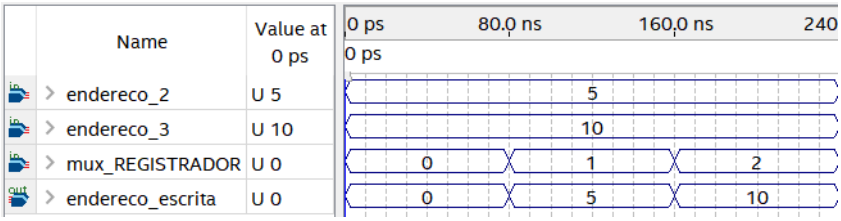
Como pode ser observado na Figura 22, inserido manualmente o valor 1 no **PC**, o **Mux_PC** varia de 0 até 3, apresentando os quatro casos de possibilidade deste multiplexador, obtendo os seguintes resultados:

- Caso 0: **novo_PC** toma o valor de **PC** e soma 1;
- Caso 1: **novo_PC** toma o valor de **PC** mais 1 e soma com o valor dado pelo **Imediato**;
- Caso 2: **novo_PC** toma o valor diretamente de **Endereco**;
- Caso 3: **novo_PC** toma o valor diretamente de **Dado_1**.

Conforme proposto pelo desenvolvimento deste multiplexador, as saídas apresentadas pelo **novo_PC** estão de acordo com cada caso dado pelo **Mux_PC**, portanto o módulo em questão apresenta individualmente comportamento esperado.

5.4.2 Multiplexador do Registrador de Escrita

Figura 23 – Simulação do Multiplexador do Registrador de Escrita



Fonte: Autor

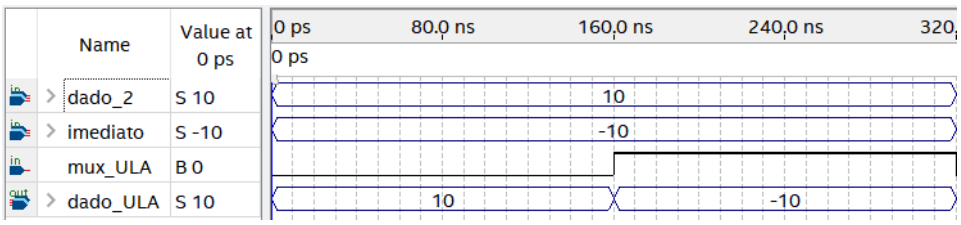
Como pode ser observado na Figura 23, inseridos manualmente os valores 5 e 10, respectivamente, nos endereços **Endereco_2** e **Endereco_3**, o **Mux_REGISTRADOR** varia de 0 até 2, apresentando os três casos de possibilidade deste multiplexador, obtendo os seguintes resultados:

- Caso 0: Caso default, **Endereco_Escrita** recebe diretamente o valor 0;
- Caso 1: **Endereco_Escrita** recebe o valor de **Endereco_2**, no caso 5;
- Caso 2: **Endereco_Escrita** recebe o valor de **Endereco_3**, no caso 10.

Conforme proposto pelo desenvolvimento deste multiplexador, as saídas apresentadas pelo **Endereco_Escrita** estão de acordo com cada caso dado pelo **Mux_REGISTRADOR**, portanto o módulo em questão apresenta individualmente comportamento esperado.

5.4.3 Multiplexador do Dado para a ULA

Figura 24 – Simulação do Multiplexador do Dado para a ULA



Fonte: Autor

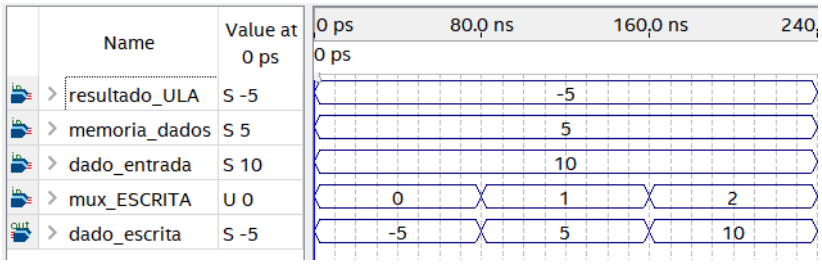
Como pode ser observado na Figura 24, inseridos manualmente os valores 10 e -10, respectivamente, no **Dado_2** e no **Imediato**, o **Mux_ULA** varia entre 0 e 1, apresentando os dois casos de possibilidade deste multiplexador, obtendo os seguintes resultados:

- Caso 0: **Dado_ULA** recebe o valor de **Dado_2**, no caso 10;
- Caso 1: **Dado_ULA** recebe o valor de **Imediato**, no caso -10.

Conforme proposto pelo desenvolvimento deste multiplexador, as saídas apresentadas pelo **Dado_ULA** estão de acordo com cada caso dado pelo **Mux_ULA**, portanto o módulo em questão apresenta individualmente comportamento esperado.

5.4.4 Multiplexador do Dado de Escrita

Figura 25 – Simulação do Multiplexador do Dado de Escrita



Fonte: Autor

Como pode ser observado na Figura 25, inseridos manualmente os valores -5, 5 e 10, respectivamente, em **Resultado_ULA**, **Memoria_Dados** e **Dado_Entrada**, o **Mux_ESCRITA** varia de 0 até 2, apresentando os três casos de possibilidade deste multiplexador, obtendo os seguintes resultados:

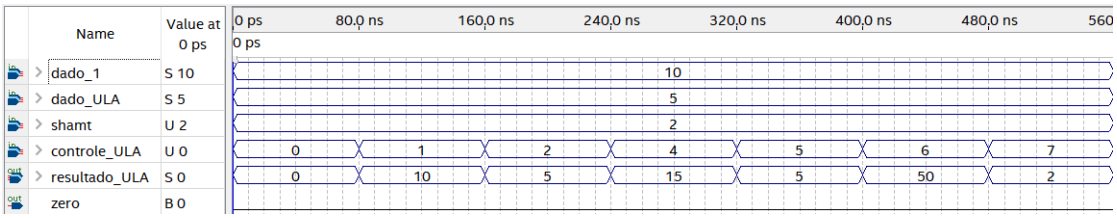
- Caso 0: **Dado_Escrita** recebe o valor de **Resultado_ULA**, no caso -5;
- Caso 1: **Dado_Escrita** recebe o valor de **Memoria_Dados**, no caso 5;
- Caso 2: **Dado_Escrita** recebe o valor de **Dado_Entrada**, no caso 10.

Conforme proposto pelo desenvolvimento deste multiplexador, as saídas apresentadas pelo **Dado_Escrita** estão de acordo com cada caso dado pelo **Mux_ESCRITA**, portanto o módulo em questão apresenta individualmente comportamento esperado.

5.5 Simulação da ULA

A fim de testar todas as operações possíveis da ULA individualmente, foram realizadas três simulações que podem ser observadas na Figura 26, Figura 27 e Figura 28.

Figura 26 – Simulação da ULA 1

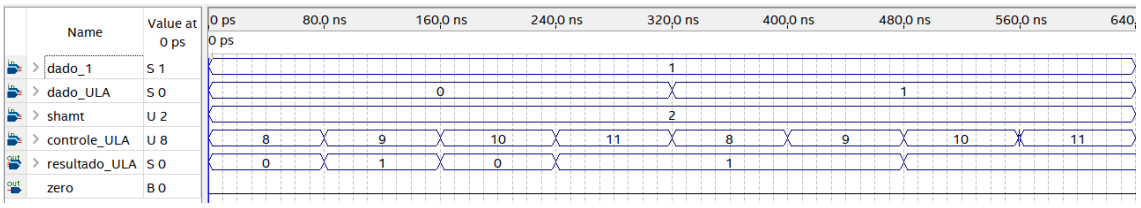


Fonte: Autor

Na primeira simulação, foram inseridos manualmente os valores 10 e 5, respectivamente, nos dados **Dado_1** e **Dado_ULA** durante toda a simulação e sendo aplicados diferentes casos utilizando o **Controle_ULA**, sendo estes:

- Caso 0 - Operação ”Default”: Neste caso, independente dos dados de entrada, **Resultado_ULA** recebe o valor 0;
- Caso 1 - Operação ”Move”: **Resultado_ULA** recebe o valor de **Dado_1**, uma vez que há movimentação de um dado para outro, no caso 10;
- Caso 2 - Operação ”Movei”: **Resultado_ULA** recebe o valor de **Dado_ULA** (no fundo, um Imediato), uma vez que há movimentação de um dado para outro, no caso 5;
- Caso 4 - Operação ”Soma com sinais”: **Resultado_ULA** recebe a soma sinalizada entre **Dado_1** e **Dado_ULA**, no caso 15;
- Caso 5 - Operação ”Subtração com sinais”: **Resultado_ULA** recebe a subtração sinalizada entre **Dado_1** e **Dado_ULA**, no caso 5;
- Caso 6 - Operação ”Multiplicação com sinais”: **Resultado_ULA** recebe a multiplicação sinalizada entre **Dado_1** e **Dado_ULA**, no caso 50;
- Caso 7 - Operação ”Divisão com sinais”: **Resultado_ULA** recebe a divisão sinalizada entre **Dado_1** e **Dado_ULA**, no caso 2.

Figura 27 – Simulação da ULA 2

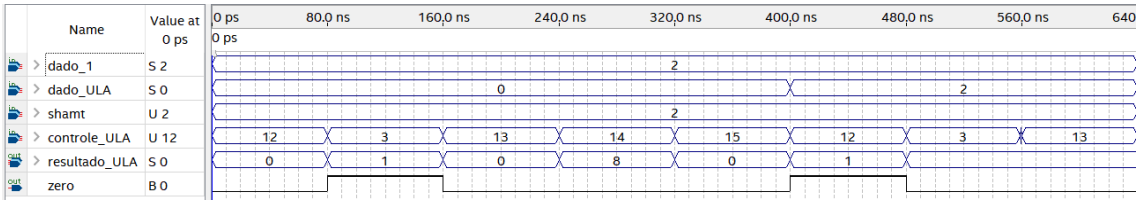


Fonte: Autor

Na segunda simulação, foram inseridos os valores 1 e 0, respectivamente, nos dados **Dado_1** e **Dado_ULA**, até o fim da primeira metade da simulação, e na segunda metade, também respectivamente, 1 e 1, sendo estes dados aplicados novamente à vários diferentes casos de operações utilizando **Controle_ULA**, sendo estes:

- Caso 8 - Operação ”AND”: **Resultado_ULA** recebe o resultado da função lógica ”AND” realizada entre **Dado_1** e **Dado_ULA**, resultando na primeira metade, 0, e na segunda metade, 1;
- Caso 9 - Operação ”OR”: **Resultado_ULA** recebe o resultado da função lógica ”OR” realizada entre **Dado_1** e **Dado_ULA**, resultando na primeira metade, 1, e na segunda metade, 1;
- Caso 10 - Operação ”NOT”: **Resultado_ULA** recebe o resultado da função lógica ”NOT” realizada no **Dado_1**, resultando na primeira metade, 0, e na segunda metade, 0;
- Caso 11 - Operação ”XOR”: **Resultado_ULA** recebe o resultado da função lógica ”XOR” realizada entre **Dado_1** e **Dado_ULA**, resultando na primeira metade, 1, e na segunda metade, 0.

Figura 28 – Simulação da ULA 3



Fonte: Autor

Na terceira e última simulação, foram inseridos manualmente os valores 2 e 0, respectivamente, nos dados **Dado_1** e **Dado_ULA** até o quinto caso apresentado nesta simulação utilizando **Controle_ULA**, para os últimos três casos (que são outras

alternativas para os três primeiros desta simulação) tivemos os valores, também respectivo aos dados, 2 e 2, sendo tais casos:

- Caso 12 - Operação ”Igual que”: Comparando **Dado_1** com **Dado_ULA**, temos que estes não são iguais na primeira tentativa, portanto **Resultado_ULA** recebe 0, na segunda tentativa os dados são iguais, portanto **Resultado_ULA** recebe 1;
- Caso 3 - Operação ”Diferente de”: Comparando **Dado_1** com **Dado_ULA**, temos que estes são diferentes na primeira tentativa, portanto **Resultado_ULA** recebe 1, na segunda tentativa os dados não são diferentes, portanto **Resultado_ULA** recebe 0;
- Caso 13 - Operação ”Menor que”: Comparando **Dado_1** com **Dado_ULA**, temos que **Dado_1** não é menor que **Dado_ULA** em ambas as tentativas, portanto **Resultado_ULA** recebe 0;
- Caso 14 - Operação ”Desloca para esquerda”: Utilizando **Shamt** para deslocar o **Dado_1** (em binário 0...0010) para a esquerda, temos que o **Resultado_ULA** recebe, em binário, o valor 0...1000, resultando no valor 8 em decimal;
- Caso 15 - Operação ”Desloca para direita”: Utilizando **Shamt** para deslocar o **Dado_1** (em binário 0...0010) para a direita, temos que o **Resultado_ULA** recebe, em binário, o valor 0...0000, resultando no valor 0 em decimal.

Terminadas todas as simulações, de todas as operações possíveis na ULA, conforme proposto pelo desenvolvimento desta, as saídas apresentadas pelo **Dado_Escrita** e pela *flag* **Zero** estão de acordo com cada caso dado pelo **Controle_ULA**, portanto o módulo em questão apresenta individualmente comportamento esperado. É interessante ressaltar que a *flag* **Zero** apenas toma valor 1 em casos positivos específicos de comparações e *branchs*, como pode ser observado, especificamente, na Figura 28.

5.6 Simulação do Processador Concatenado

Uma vez concatenado todos os módulos individuais em um esquemático ”.bdf”, que pode ser visto na Figura 16, concluindo no processador inteiro, o seguinte algoritmo, que pode ser visto brevemente na subseção 4.6.1, que é uma espécie de Fibonacci, foi aplicado ao processador e enquanto era feita uma simulação, contida na Figura 29:

1. Tipo R - ”movei”: R[5] = 5:

   32’b000011_00000_00101_0000000000000101

2. Tipo R - ”movei”: R[1] = 1:

   32’b000011_00000_00001_0000000000000001

3. Tipo R - ”movei”: R[2] = 1:

   32’b000011_00000_00010_0000000000000001

4. Tipo R - ”add”: R[3] = R[1] + R[2]:

   32’b000000_00001_00010_00011_00000_000100

5. Tipo R - ”move”: R[1] = R[2]:

   32’b000000_00010_00000_00001_00000_000011

6. Tipo R - ”move”: R[2] = R[3]:

   32’b000000_00011_00000_00010_00000_000011

7. Tipo R - ”out”: *output* = R[1]:

   32’b000000_00001_00000_00000_00000_000010

8. Tipo R - ”out”: *output* = R[2]:

   32’b000000_00010_00000_00000_00000_000010

9. Tipo I - ”store”: M[R[0] + 2] = R[3]:

   32’b000010_00000_00011_0000000000000010

10. Tipo I - ”load”: R[4] = M[R[0] + 2]:

    32’b000001_00000_00100_0000000000000010

11. Tipo R - ”out”: *output* = R[4]:

    32’b000000_00100_00000_00000_00000_000010

12. Tipo R - ”out”: *output* = R[5]:

    32’b000000_00101_00000_00000_00000_000010

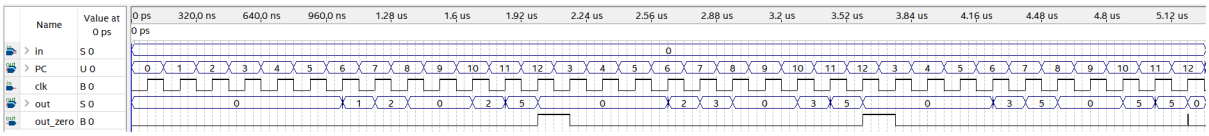
13. Tipo I - ”bne”: (R[4]!=R[5]), PC = PC + 1 + (-10):

    32’b001101_00100_00101_1111111111110110

14. Tipo J - ”jump”: PC = 13:

    32’b010000_00000000000000000000001101

Figura 29 – Simulação do Processador Fibonacci



Fonte: Autor

Com base nos ***outputs*** disponíveis segundo a simulação e considerando que o único *input* dado manualmente foi o algoritmo (como um usuário utilizaria normalmente), é dado que o resultado final do processador concatenado, ou seja, completo está conforme proposto durante seu desenvolvimento, portanto este processador apresenta o comportamento esperado.

6 Considerações Finais

Como consideração final do projeto de arquitetar e desenvolver um processador do zero em lógica programável para que pudesse ser interagido com programação exterior em baixo nível e utilizar um kit FPGA como interface de E/S, tivemos no percurso que, durante a primeira etapa esse pareceu fácil de se esquematizar e elaborar segundo a arquitetura base (MIPS32), uma vez que esta tinha muito material disponível em livros, sites e afins, assim como havia a disponibilidade de um relatório completo sobre o assunto para se basear, tornando a primeira etapa bastante teórica e repetitiva a certo ponto, porém sem dificuldades; durante a segunda etapa, relatada neste documento, houve a confirmação do sucesso em implementar o processador individualmente e concatenar tais módulos a fim de se obter um processador completo e funcional, provado pela realização de um algoritmo sofisticado a certo ponto, o ”Fibonacci”.

Apesar de algumas dificuldades no decorrer da implementação como a instabilidade e dificuldade de decisão em relação às subidas e descidas do *clock*, ou na exaustiva repetição de traduzir todas as instruções para a Unidade de Controle e, posteriormente, a ULA, o projeto se demonstrou satisfatório e muito cativante em toda sua magnitude.

Talvez um único ponto negativo em relação ao projeto num todo, justificável pelo período em que este foi produzido, seja a falta do kit FPGA citado para testes mais profundos e palpáveis de fato.

Terminado o projeto completamente em relação à esta matéria, os próximos passos contam com a implementação de mais instruções para manter o processador atualizado e com uma variedade maior de opções para que as próximas matérias que o utilize tenham maior facilidade de serem aproveitadas ao invés de serem pegas de surpresa com a necessidade de algo a ser implementado, assim como contam com a otimização e organização desse projeto no geral, incluindo uma melhor implementação da interface de E/S baseada e pensada na interação com um kit FPGA em mãos.

Referências

1 HENNESY, J. L. *Arquitetura de Computadores: Uma Abordagem Quantitativa*. 5ª edição. ed. Rio de Janeiro: Elsevier, 2014. Citado 3 vezes nas páginas 7, 15 e 16.

2 STALLINGS, W. *Arquitetura e Organização de Computadores*. 8ª edição. ed. São Paulo: Pearson Pratice Hall, 2010. Citado 3 vezes nas páginas 11, 12 e 14.

3 FLOYD, T. L. *Sistemas Digitais: Fundamentos e Aplicações*. 9ª edição. ed. Porto Alegre: Bookman, 2007. Citado 3 vezes nas páginas 17, 18 e 19.

Apêndices

APÊNDICE A – ULA

Código do arquivo ULA.v

```
1  module ULA (dado_1, dado_ULA, shamt, controle_ULA, resultado_ULA, zero);
2
3          input [31:0]dado_1;                                     //Entrada do DADO 1
4          input [31:0]dado_ULA;                                 //Entrada do DADO 2 ou IMEDIATO
5          input [4:0]shamt;                                                 //Entrada do
               Shamt
6          input [3:0]controle_ULA;                              //Entrada do CONTROLADOR
7          output reg [31:0]resultado_ULA; //Saida do resultado da ULA
8          output reg zero;                                                  //Flag zero
9
10         initial
11                 begin
12                         zero = 0;
13                         resultado_ULA = 0;
14                 end
15
16         always @(*)
17         begin
18                 case(controle_ULA)
19                         4'b0001:                //Move o DADO_1, ou seja, R[d] <- R[s] -
                               move
20                             begin
21                                     resultado_ULA = dado_1;
22                                     zero = 0;
23                             end
24
25                         4'b0010:                //Move o IMEDIATO, ou seja, R[t] <- IMD -
                                movei
26                             begin
27                                     resultado_ULA = dado_ULA;
28                                     zero = 0;
29                             end
30
31                         4'b0100:                //Soma com sinais - add, addi, load,
                               store
32                             begin
33                                     resultado_ULA = $signed(dado_1) + $signed(
                                           dado_ULA);
34                                     zero = 0;
35                             end
36
37                         4'b0101:                //Subtracao com sinais - sub, subi
38                             begin
39                                     resultado_ULA = $signed(dado_1) - $signed(
                                           dado_ULA);
40                                     zero = 0;
41                             end
42
43                         4'b0110:                //Multiplicacao com sinais - mult
44                             begin
45                                     resultado_ULA = $signed(dado_1) * $signed(
                                           dado_ULA);
```

```
46                                      zero = 0;
47                              end
48
49                      4'b0111:                //Divisao com sinais - div
50                              begin
51                                      if(dado_ULA == 32'd0) resultado_ULA = 32'd0;
                                            //Divis o por 0
52                                      else resultado_ULA = $signed(dado_1) / $signed(
                                            dado_ULA);
53                                      zero = 0;
54                              end
55
56                      4'b1000:                //AND entre os dados - and, andi
57                              begin
58                                      resultado_ULA = (dado_1 && dado_ULA);
59                                      zero = 0;
60                              end
61
62                      4'b1001:                //OR entre os dados - or, ori
63                              begin
64                                      resultado_ULA = (dado_1 || dado_ULA);
65                                      zero = 0;
66                              end
67
68                      4'b1010:                //NOT do DADO_1 - not
69                              begin
70                                      resultado_ULA = !(dado_1);
71                                      zero = 0;
72                              end
73
74                      4'b1011:                //XOR entre os dados - xor
75                              begin
76                                      resultado_ULA = (dado_1 ^ dado_ULA);
77                                      zero = 0;
78                              end
79
80                      4'b1100:                //Se DADO_1 for igual a DADO_ULA, 1, caso
                            contr rio, 0 - beq, set
81                              begin
82                                      if($signed(dado_1) == $signed(dado_ULA)) begin
83                                                      resultado_ULA = 32'd1;
84                                                      zero = 1;
85                                      end else begin
86                                                      resultado_ULA = 32'd0;
87                                                      zero = 0;
88                                      end
89                              end
90
91                      4'b0011:                //Se DADO_1 for diferente a DADO_ULA, 1,
                            caso contr rio, 0 - bne
92                              begin
93                                      if($signed(dado_1) != $signed(dado_ULA)) begin
94                                                      resultado_ULA = 32'd1;
95                                                      zero = 1;
96                                      end else begin
97                                                      resultado_ULA = 32'd0;
98                                                      zero = 0;
99                                      end
100                             end
101
```

```
102                         4'b1101:                //Se DADO_1 for menor que DADO_ULA, 1,
                                caso contrario, 0 - slt
103                             begin
104                                             if($signed(dado_1) < $signed(dado_ULA))
                                                    begin
105                                                         resultado_ULA = 32'd1;
106                                                         zero = 1;
107                                             end else begin
108                                                         resultado_ULA = 32'd0;
109                                                         zero = 0;
110                                             end
111                                     end
112
113                         4'b1110:                //Desloca DADO_1 para esquerda em SHAMT
                                bits - sll
114                             begin
115                                     resultado_ULA = (dado_1 <<< shamt);
116                                     zero = 0;
117                             end
118
119                         4'b1111:                //Desloca DADO_1 para direita em SHAMT
                                bits - srl
120                             begin
121                                     resultado_ULA = (dado_1 >>> shamt);
122                                     zero = 0;
123                             end
124
125                         default:
126                             begin
127                                     resultado_ULA = 32'd0;
128                                     zero = 0;
129                             end
130                 endcase
131         end
132 endmodule
```

APÊNDICE B – UC

Código do arquivo UC.v

```
1  module UC (instrucao, zero, controle_entrada, controle_saida, controle_ULA,
       controle_memoriaDADOS, controle_REGISTRADOR, mux_PC, mux_ULA, mux_REGISTRADOR,
       mux_ESCRITA);
2
3          input [31:0]instrucao;                                    //Entrada da INSTRUCAO
4          input zero;
               //Entrada da flag ZERO
5          output reg controle_entrada;                    //Saida do CONTROLADOR do input
6          output reg controle_saida;                               //Saida do CONTROLADOR do
               output
7          output reg [3:0]controle_ULA;                   //Saida do CONTROLADOR da ULA
8          output reg controle_memoriaDADOS;       //Saida do CONTROLADOR da memoria de
               dados
9          output reg controle_REGISTRADOR;                //Saida do CONTROLADOR do banco
               de registradores
10         output reg [1:0]mux_PC;                                  //Saida do CONTROLADOR do
               mult do PC
11         output reg mux_ULA;                                                //Saida do
               CONTROLADOR do mult da ULA
12         output reg [1:0]mux_REGISTRADOR;                //Saida do CONTROLADOR do mult do
               endereco de escrita
13         output reg [1:0]mux_ESCRITA;                    //Saida do CONTROLADOR do mult do
               dado de escrita
14
15         initial begin
16                 controle_entrada = 0;
17                 controle_saida = 0;
18                 controle_ULA = 0;
19                 controle_memoriaDADOS = 0;
20                 controle_REGISTRADOR = 0;
21                 mux_PC = 0;
22                 mux_ULA = 0;
23                 mux_REGISTRADOR = 0;
24                 mux_ESCRITA = 0;
25         end
26
27         always @(instrucao)
28         begin
29                 if(instrucao[31:26]==6'b000000 && instrucao[5:0]==6'b000001)
                       controle_entrada = 1'b1;
30                 else controle_entrada = 1'b0;   //Define se h   ou nao input
31
32                 if(instrucao[31:26]==6'b000000 && instrucao[5:0]==6'b000010)
                       controle_saida = 1'b1;
33                 else controle_saida = 1'b0;     //Define se h   ou nao output
34         end
35
36         always @(*)
37         case(instrucao[31:26])
38         6'b000000:                                                         //Especifico do
               Tipo R, o "real" OPCODE est  na FUNCT logo abaixo
39                 case(instrucao[5:0])
```

```
40              6'b000001:                              //Comando: in
41                      begin
42                              controle_ULA = 0;
                                    //Codigo da ULA: R[d] toma INPUT
43                              controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
44                              controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
45                              mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
46                              mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
47                              mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
48                              mux_ESCRITA = 2;
                                    //INPUT para escrever no banco de registradores
49                      end
50              6'b000010:                              //Comando: out
51                      begin
52                              controle_ULA = 0;
                                    //Codigo da ULA: OUTPUT toma R[s]
53                              controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
54                              controle_REGISTRADOR = 0;               //NAO escrevendo
                                    no banco de registrador
55                              mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
56                              mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
57                              mux_REGISTRADOR = 0;                            //Sem
                                    endereco para escrever, ou seja, "registrador[0]" ou
                                    "R[0]"
58                              mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
59                      end
60              6'b000011:                              //Comando: move
61                      begin
62                              controle_ULA = 4'b0001;                 //Codigo da ULA:
                                    Mover Registrador
63                              controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
64                              controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
65                              mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
66                              mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
67                              mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
68                              mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
69                      end
70              6'b000100:                              //Comando: add
71                      begin
72                              controle_ULA = 4'b0100;                 //Codigo da ULA:
                                    Soma
```

```
73                              controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
74                              controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
75                              mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
76                              mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
77                              mux_REGISTRADOR = 2;                                //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
78                              mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
79                      end
80                  6'b000101:                              //Comando: sub
81                      begin
82                              controle_ULA = 4'b0101;                 //Codigo da ULA:
                                    Subtracao
83                              controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
84                              controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
85                              mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
86                              mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
87                              mux_REGISTRADOR = 2;                                //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
88                              mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
89                      end
90                  6'b000110:                              //Comando: mult
91                      begin
92                              controle_ULA = 4'b0110;                 //Codigo da ULA:
                                    Multiplicacao
93                              controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
94                              controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
95                              mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
96                              mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
97                              mux_REGISTRADOR = 2;                                //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
98                              mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
99                      end
100                 6'b000111:                              //Comando: div
101                     begin
102                             controle_ULA = 4'b0111;                 //Codigo da ULA:
                                    Divisao
103                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
```

```
104                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
105                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
106                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
107                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
108                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
109                     end
110             6'b001000:                              //Comando: and
111                     begin
112                             controle_ULA = 4'b1000;                 //Codigo da ULA:
                                    AND
113                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
114                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
115                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
116                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
117                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
118                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
119                     end
120             6'b001001:                              //Comando: or
121                     begin
122                             controle_ULA = 4'b1001;                 //Codigo da ULA:
                                    OR
123                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
124                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
125                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
126                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
127                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
128                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
129                     end
130             6'b001010:                              //Comando: not
131                     begin
132                             controle_ULA = 4'b1010;                 //Codigo da ULA:
                                    NOT
133                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
134                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
```

```
135                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
136                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
137                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
138                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
139                     end
140             6'b001011:                              //Comando: xor
141                     begin
142                             controle_ULA = 4'b1011;                 //Codigo da ULA:
                                    XOR
143                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
144                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
145                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
146                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
147                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
148                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
149                     end
150             6'b001100:                              //Comando: set (Igual que)
151                     begin
152                             controle_ULA = 4'b1100;                 //Codigo da ULA:
                                    DADO 1 == DADO 2
153                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
154                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
155                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
156                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
157                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
158                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
159                     end
160             6'b001101:                              //Comando: slt (Menor que)
161                     begin
162                             controle_ULA = 4'b1101;                 //Codigo da ULA:
                                    DADO 1 < DADO 2
163                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
164                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
165                             mux_PC = 0;
                                            //NAO salta, PC toma PC + 1
```

```
166                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
167                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
168                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
169                     end
170                 6'b001110:                              //Comando: sll (Desloca para
                        esquerda)
171                     begin
172                             controle_ULA = 4'b1110;                 //Codigo da ULA:
                                    Desloca DADO 1 p/ esquerda em SHAMT bits
173                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
174                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
175                             mux_PC = 0;
                                        //NAO salta, PC toma PC + 1
176                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
177                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
178                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
179                     end
180                 6'b001111:                              //Comando: srl (Desloca para
                        direita)
181                     begin
182                             controle_ULA = 4'b1111;                 //Codigo da ULA:
                                    Desloca DADO 1 p/ direita em SHAMT bits
183                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
184                             controle_REGISTRADOR = 1;               //SIM escrevendo
                                    no banco de registrador
185                             mux_PC = 0;
                                        //NAO salta, PC toma PC + 1
186                             mux_ULA = 0;
                                    //DADO_ULA toma DADO 2 na ULA
187                             mux_REGISTRADOR = 2;                            //
                                    Utilizando ENDERECO 3 para escrever, ou seja, "
                                    registrador[endereco_3]" ou "R[d]"
188                             mux_ESCRITA = 0;
                                    //Resultado da ULA para escrever no banco de
                                    registradores
189                     end
190                 6'b010000:                              //Comando: jr (Jump to
                        REGISTRADOR)
191                     begin
192                             controle_ULA = 0;
                                    //Codigo da ULA: Default
193                             controle_memoriaDADOS = 0;              //NAO escrevendo
                                    na memoria de dados
194                             controle_REGISTRADOR = 0;               //NAO escrevendo
                                    no banco de registrador
195                             mux_PC = 3;
                                        //SIM salta, PC toma DADO existente em "
```

```
                                registrador[endereco_leitura1]" ou "R[s]"
196                         mux_ULA = 0;
                                //DADO_ULA toma DADO 2 na ULA
197                         mux_REGISTRADOR = 0;                            //Sem
                                endereco para escrever, ou seja, "registrador[0]" ou
                                "R[0]"
198                         mux_ESCRITA = 0;
                                //Resultado da ULA para escrever no banco de
                                registradores
199                 end
200             default:                                        //Blank - Normalmente
                    comeco do c digo, nao realiza nenhuma funcao
201                 begin
202                         controle_ULA = 0;
                                //Codigo da ULA: Default
203                         controle_memoriaDADOS = 0;              //NAO escrevendo
                                na memoria de dados
204                         controle_REGISTRADOR = 0;               //NAO escrevendo
                                no banco de registrador
205                         mux_PC = 0;
                                    //NAO salta, PC toma PC + 1
206                         mux_ULA = 0;
                                //DADO_ULA toma DADO 2 na ULA
207                         mux_REGISTRADOR = 0;                            //Sem
                                endereco para escrever, ou seja, "registrador[0]" ou
                                "R[0]"
208                         mux_ESCRITA = 0;
                                //Resultado da ULA para escrever no banco de
                                registradores
209                 end
210             endcase
211         6'b000001:                                                  //Comando: load
212             begin
213                     controle_ULA = 4'b0100;                 //Codigo da ULA: Soma
214                     controle_memoriaDADOS = 0;              //NAO escrevendo na
                            memoria de dados
215                     controle_REGISTRADOR = 1;               //SIM escrevendo no banco
                             de registrador
216                     mux_PC = 0;
                            //NAO salta, PC toma PC + 1
217                     mux_ULA = 1;                                            //
                            DADO_ULA toma IMEDIATO na ULA
218                     mux_REGISTRADOR = 1;                            //Utilizando
                            ENDERECO 2 para escrever, ou seja, "registrador[endereco_2]"
                            ou "R[t]"
219                     mux_ESCRITA = 1;                                        //Dado da
                             Memoria para escrever no banco de registradores
220             end
221         6'b000010:                                                  //Comando: store
222             begin
223                     controle_ULA = 4'b0100;                 //Codigo da ULA: Soma
224                     controle_memoriaDADOS = 1;              //SIM escrevendo na
                            memoria de dados
225                     controle_REGISTRADOR = 0;               //NAO escrevendo no banco
                             de registrador
226                     mux_PC = 0;
                            //NAO salta, PC toma PC + 1
227                     mux_ULA = 1;                                            //
                            DADO_ULA toma IMEDIATO na ULA
```

```
228                     mux_REGISTRADOR = 0;                            //Sem endereco
                            para escrever, ou seja, "registrador[0]" ou "R[0]"
229                     mux_ESCRITA = 0;                                        //
                            Resultado da ULA para escrever no banco de registradores
230             end
231         6'b000011:                                                  //Comando: movei
232             begin
233                     controle_ULA = 4'b0010;                 //Codigo da ULA: Mover
                            IMEDIATO
234                     controle_memoriaDADOS = 0;              //NAO escrevendo na
                            memoria de dados
235                     controle_REGISTRADOR = 1;               //SIM escrevendo no banco
                             de registrador
236                     mux_PC = 0;
                            //NAO salta, PC toma PC + 1
237                     mux_ULA = 1;                                            //
                            DADO_ULA toma IMEDIATO na ULA
238                     mux_REGISTRADOR = 1;                            //Utilizando
                            ENDERECO 2 para escrever, ou seja, "registrador[endereco_2]"
                            ou "R[t]"
239                     mux_ESCRITA = 0;                                        //
                            Resultado da ULA para escrever no banco de registradores
240             end
241         6'b000100:                                                  //Comando: addi
242             begin
243                     controle_ULA = 4'b0100;                 //Codigo da ULA: Soma
244                     controle_memoriaDADOS = 0;              //NAO escrevendo na
                            memoria de dados
245                     controle_REGISTRADOR = 1;               //SIM escrevendo no banco
                             de registrador
246                     mux_PC = 0;
                            //NAO salta, PC toma PC + 1
247                     mux_ULA = 1;                                            //
                            DADO_ULA toma IMEDIATO na ULA
248                     mux_REGISTRADOR = 1;                            //Utilizando
                            ENDERECO 2 para escrever, ou seja, "registrador[endereco_2]"
                            ou "R[t]"
249                     mux_ESCRITA = 0;                                        //
                            Resultado da ULA para escrever no banco de registradores
250             end
251         6'b000101:                                                  //Comando: subi
252             begin
253                     controle_ULA = 4'b0101;                 //Codigo da ULA:
                            Subtracao
254                     controle_memoriaDADOS = 0;              //NAO escrevendo na
                            memoria de dados
255                     controle_REGISTRADOR = 1;               //SIM escrevendo no banco
                             de registrador
256                     mux_PC = 0;
                            //NAO salta, PC toma PC + 1
257                     mux_ULA = 1;                                            //
                            DADO_ULA toma IMEDIATO na ULA
258                     mux_REGISTRADOR = 1;                            //Utilizando
                            ENDERECO 2 para escrever, ou seja, "registrador[endereco_2]"
                            ou "R[t]"
259                     mux_ESCRITA = 0;                                        //
                            Resultado da ULA para escrever no banco de registradores
260             end
261         6'b001000:                                                  //Comando: andi
262             begin
```

```
263                         controle_ULA = 4'b1000;                 //Codigo da ULA: AND
264                         controle_memoriaDADOS = 0;              //NAO escrevendo na
                                memoria de dados
265                         controle_REGISTRADOR = 1;               //SIM escrevendo no banco
                                 de registrador
266                         mux_PC = 0;
                                //NAO salta, PC toma PC + 1
267                         mux_ULA = 1;                                            //
                                DADO_ULA toma IMEDIATO na ULA
268                         mux_REGISTRADOR = 1;                            //Utilizando
                                ENDERECO 2 para escrever, ou seja, "registrador[endereco_2]"
                                ou "R[t]"
269                         mux_ESCRITA = 0;                                        //
                                Resultado da ULA para escrever no banco de registradores
270                 end
271         6'b001001:                                                      //Comando: ori
272                 begin
273                         controle_ULA = 4'b1001;                 //Codigo da ULA: OR
274                         controle_memoriaDADOS = 0;              //NAO escrevendo na
                                memoria de dados
275                         controle_REGISTRADOR = 1;               //SIM escrevendo no banco
                                 de registrador
276                         mux_PC = 0;
                                //NAO salta, PC toma PC + 1
277                         mux_ULA = 1;                                            //
                                DADO_ULA toma IMEDIATO na ULA
278                         mux_REGISTRADOR = 1;                            //Utilizando
                                ENDERECO 2 para escrever, ou seja, "registrador[endereco_2]"
                                ou "R[t]"
279                         mux_ESCRITA = 0;                                        //
                                Resultado da ULA para escrever no banco de registradores
280                 end
281         6'b001100:                                                      //Comando: beq
282                 begin
283                         controle_ULA = 4'b1100;                 //Codigo da ULA: DADO 1
                                == DADO 2, PC toma PC + 1 + IMD
284                         controle_memoriaDADOS = 0;              //NAO escrevendo na
                                memoria de dados
285                         controle_REGISTRADOR = 0;               //NAO escrevendo no banco
                                 de registrador
286                         mux_ULA = 0;                                            //
                                DADO_ULA toma DADO 2 na ULA
287                         mux_REGISTRADOR = 0;                            //Sem endereco
                                para escrever, ou seja, "registrador[0]" ou "R[0]"
288                         mux_ESCRITA = 0;                                        //
                                Resultado da ULA para escrever no banco de registradores
289                         if(zero)                                mux_PC = 1;
                                //SIM salta, PC toma PC + 1
290                         else if(!zero)          mux_PC = 0;             //NAO salta, PC
                                toma PC + 1 + IMEDIATO
291                 end
292         6'b001101:                                                      //Comando: bne
293                 begin
294                         controle_ULA = 4'b0011;                 //Codigo da ULA: DADO 1
                                != DADO 2, PC toma PC + 1 + IMD
295                         controle_memoriaDADOS = 0;              //NAO escrevendo na
                                memoria de dados
296                         controle_REGISTRADOR = 0;               //NAO escrevendo no banco
                                 de registrador
```

```
297                         mux_ULA = 0;                                                //
                                DADO_ULA toma DADO 2 na ULA
298                         mux_REGISTRADOR = 0;                                //Sem endereco
                                para escrever, ou seja, "registrador[0]" ou "R[0]"
299                         mux_ESCRITA = 0;                                            //
                                Resultado da ULA para escrever no banco de registradores
300                         if(zero)                                    mux_PC = 1;
                                //SIM salta, PC toma PC + 1
301                         else if(!zero)          mux_PC = 0;             //NAO salta, PC
                                toma PC + 1 + IMEDIATO
302                 end
303         6'b010000:                                                          //Comando: j (
                Jump to ENDERECO)
304                 begin
305                         controle_ULA = 0;                                           //Codigo
                                da ULA: Default
306                         controle_memoriaDADOS = 0;              //NAO escrevendo na
                                memoria de dados
307                         controle_REGISTRADOR = 0;               //NAO escrevendo no banco
                                 de registrador
308                         mux_PC = 2;
                                //SIM salta, PC toma o IMEDIATO dado por ENDERECO
309                         mux_ULA = 0;                                                //
                                DADO_ULA toma DADO 2 na ULA
310                         mux_REGISTRADOR = 0;                                //Sem endereco
                                para escrever, ou seja, "registrador[0]" ou "R[0]"
311                         mux_ESCRITA = 0;                                            //
                                Resultado da ULA para escrever no banco de registradores
312                 end
313         default:                                                                    //Blank -
                 Normalmente comeco do c digo, nao realiza nenhuma funcao
314                 begin
315                         controle_ULA = 0;                                           //Codigo
                                da ULA: Default
316                         controle_memoriaDADOS = 0;              //NAO escrevendo na
                                memoria de dados
317                         controle_REGISTRADOR = 0;               //NAO escrevendo no banco
                                 de registrador
318                         mux_PC = 0;
                                //NAO salta, PC toma PC + 1
319                         mux_ULA = 0;                                                //
                                DADO_ULA toma DADO 2 na ULA
320                         mux_REGISTRADOR = 0;                                //Sem endereco
                                para escrever, ou seja, "registrador[0]" ou "R[0]"
321                         mux_ESCRITA = 0;                                            //
                                Resultado da ULA para escrever no banco de registradores
322                 end
323         endcase
324 endmodule
```