Pedro Spoljaric Gomes

Projeto e Desenvolvimento de um Sistema Computacional Baseado em Arquitetura RISC - Ponto de Checagem 4 (final)

São José dos Campos - Brasil

julho de 2019

Pedro Spoljaric Gomes

Projeto e Desenvolvimento de um Sistema Computacional Baseado em Arquitetura RISC - Ponto de Checagem 4 (final)

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. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

julho de 2019

Resumo

Este trabalho descreve o projeto e desenvolvimento de um processador baseado em arquitetura RISC que tem como principal objetivo executar os algoritmos básicos da disciplina de Lógica de Programação. O conjunto de instruções foi definido previamente e as unidades funcionais integradas através de um esquemático da arquitetura desenvolvida. A implementação da estrutura foi desenvolvida na linguagem verilog de descrição de hardware, foram geradas formas de onda no software Quartus Prime e realizados testes no dispositivo FPGA mostrando resultados condizentes com o objetivo do projeto.

Palavras-chaves: FPGA, Arquitetura RISC, verilog, sistema computacional, processador.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A importância da arquitetura de um computador se consolida já no processamento mais básico realizado nele, onde valores de tensão em transistores são convertidos em *bits* e interpretados pelo processador como dados de entrada que devem ser executados. A arquitetura é o modelo que define como esses dados serão interpretados, o que será feito com eles e para onde vão, como um simples comando que realiza a soma entre dois números e armazena o resultado. De acordo com Alan Clements em *Principles of Computer Hardware*, "A arquitetura descreve a organização interna de um computador de um jeito abstrato; isto é, ela define as capacidades de um computador e seu modelo de programação. Você pode ter dois computadores que foram construídos de maneiras diferentes com tecnologias diferentes, mas com a mesma arquitetura" (1, p. 1, tradução nossa).

O primeiro passo para projetar o sistema computacional foi a definição dos conceitos básicos de *design* do projeto, como os componentes que serão usados e o modo como os comandos serão executados. Ao término dessa etapa do projeto, todos esses conceitos da arquitetura estavam definidos, prevendo que ela seja capaz de executar algoritmos básicos. Além das definições iniciais, nessa etapa foram apresentados todos os tipos de instruções que serão suportados pelo processador e seus caminhos de execução (*datapaths*), através de um esquemático construído para descrever e ilustrar todo o processo.

Em seguida, foi apresentado todo o processo de implementação dos componentes que fazem parte das unidades de processamento e de controle do sistema computacional, assim como a conexão entre eles.

Finalmente, nessa etapa do projeto o processador foi finalizado com os módulos de entrada e saída implementados, sendo capaz de executar os algoritmos vistos em Lógica de Programação.

2 Objetivos

2.1 Geral

Projetar uma arquitetura computacional completa em lógica programável, definindo todos os componentes do processador, tipos de dados, modos de endereçamento e conjunto de instruções, a fim de que seja possível executar algoritmos básicos a partir dela.

2.2 Específicos

- Definir os componentes que farão parte do processador.
- Definir os tipos de dados que serão armazenados.
- Definir os modos de endereçamento de dados.
- Definir todos os tipos e subtipos de instrução, seu formato e modo que serão executadas pelo processador (*datapath*);
- Construir um esquemático que descreve todas essas definições;
- Implementar em verilog os blocos das unidades de processamento e de controle;
- Implementar em verilog a integração entre os blocos das unidades de processamento e de controle;
- Simular os blocos das unidades de processamento e de controle;
- Simular a integração dos blocos das unidades de processamento e de controle;
- Implementar em verilog os módulos de entrada e saída;
- Simular os módulos de entrada e saída;
- Integrar o sistema com o dispositivo FPGA e realizar os testes necessários.

3 Fundamentação Teórica

Para projetar um processador é necessário estar a par de alguns conceitos muito importantes sobre Arquitetura de Computadores, como por exemplo, os principais componentes que formarão este processador: unidade de controle, unidade de processamento e banco de registradores; assim como a definição do conjunto de instruções e dos modos de endereçamento. Todos esses conceitos serão abordados nos tópicos a seguir.

3.1 Arquitetura

De maneira geral, arquitetura de um computador é a definição de seus conceitos e fundamentos de mais baixo nível operacional , ou seja, é a base de todo o projeto de um sistema computacional, incluindo a escolha das peças físicas que serão utilizadas, a organização dos componentes, a linguagem que será interpretada pelo computador, o modo como o computador os interpreta, entre muitas outras coisas que consolidam a base estrutural e funcional do sistema. (2)

3.2 Processador

Segundo o Prof. Jean Galdino (3), o processador, ou também conhecido como CPU, é um circuito integrado programável que realiza as funções de cálculo e tomada de decisão de um computador, sendo o componente mais complexo e mais importante de um dispositivo. Sua estrutura básica contém uma unidade de controle, uma unidade de processamento, e um banco de registradores.

3.2.1 Unidade de Processamento

A unidade de processamento, ou unidade lógica e aritmética (ULA), é responsável por executar os programas, instruções lógicas, matemáticas, desvio, entre outras. Uma operação específica da ULA é determinada por um código binário específico colocado nas entradas de seleção de funções (4).

3.2.2 Unidade de Controle

A unidade de controle realiza a tarefa de controle das ações a serem realizadas pelo processador, podendo acessar, decodificar e executar as instruções sucessivas de um programa armazenado na memória. Segundo o Prof. Raimundo Filho (5), ela gera e

gerencia os sinais de controle necessários para sincronizar operações, bem como o fluxo de instruções de um programa e dados dentro e fora da ULA.

3.2.3 Bando de Registradores

Um processador contém um banco de registradores, que são circuitos capazes de receber informações, guardá-las e transferi-las na direção de algum dispositivo de controle. Segundo o Prof. Raimundo Filho (5), os registradores são algumas posições de memória com as mesmas características das palavras da memória principal.

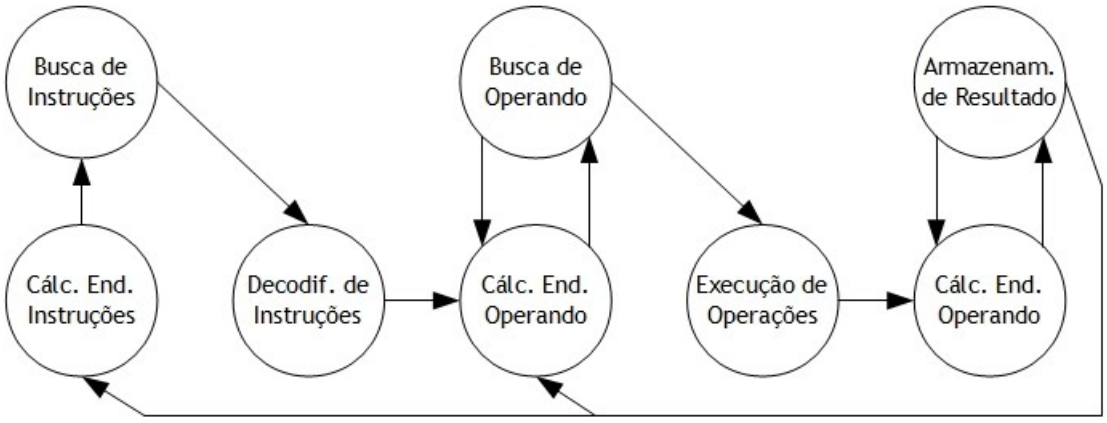
As informações, ou sequência de *bits*, guardadas nos registradores podem ser deslocadas para a direita ou para a esquerda dentro do registrador ou podem ser transferidas entre dois registradores, podendo guardar informações temporariamente, enquanto outra parte do processador analisa os dados.

3.3 Conjunto de instruções

Uma instrução é a interface entre o *hardware* e o *software*, ou seja, é a parte visível para o programador, sendo formada por campos que guardam os dados necessários para a execução da instrução, geralmente apresentados como: Código de Operação e Campos de Endereço (6).

Os Códigos de Operação, ou opcodes, indicam a operação que vai ser executada, tanto quanto o modo de endereçamento de operandos utilizado. E os Campos de Endereço indicam o local de cada operando, podendo variar de acordo com a necessidade da operação. Quando uma instrução é apresentada ao processador, ele ativa cada um de seus componentes internos para realizar uma parte da execução desta instrução. Esse processo é demonstrado na Figura 1.

Figura 1 – Diagrama do ciclo de instrução



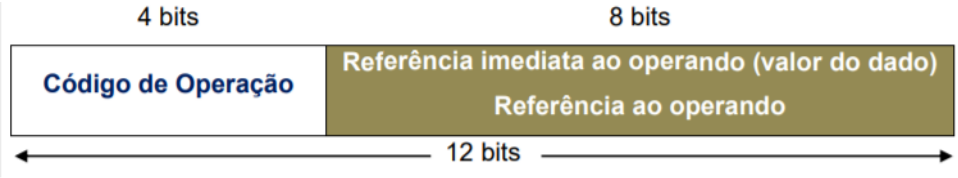
Fonte: Caderno *Geek* (6)

3.4 Modos de endereçamento

Segundo William Stallings (7), "o campo ou os campos de endereço em um formato de instrução típico são relativamente pequenos. Para que pudéssemos referenciar um grande intervalo de locais da memória principal - ou, em alguns sistemas, da memória virtual -, uma variedade de técnicas de endereçamento foi empregada." Os modos de endereçamento que serão utilizados são: imediato, direto, por registrador e por registrador base.

O modo de endereçamento imediato é o método mais simples e rápido de obter um dado, pois indica seu próprio valor no campo operando da instrução, em vez de buscá-lo na memória, conforme Figura 2.

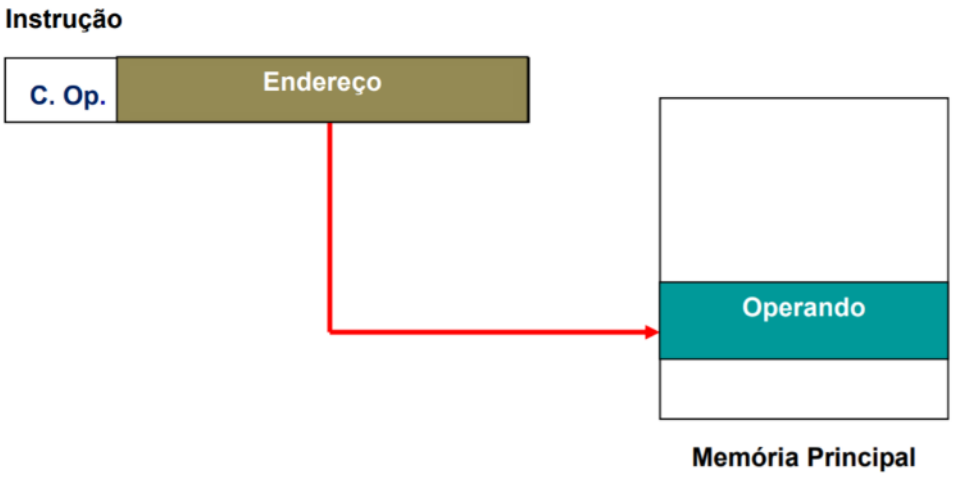
Figura 2 – Instrução do modo de endereçamento imediato



Fonte: Alexandre Lucas Chichosz et al (8)

O modo de endereçamento direto mostrado na Figura 3, de acordo com Stallings (7), é um método no qual o campo de endereço da instrução contém o endereço efetivo do operando, que por sua vez encontra-se na memória principal. Esse endereçamento é também um modo simples de acesso, pois requer apenas uma referência a memória principal para buscar o dado, sendo, porém, mais lento que o modo imediato por fazer referência à memória (9).

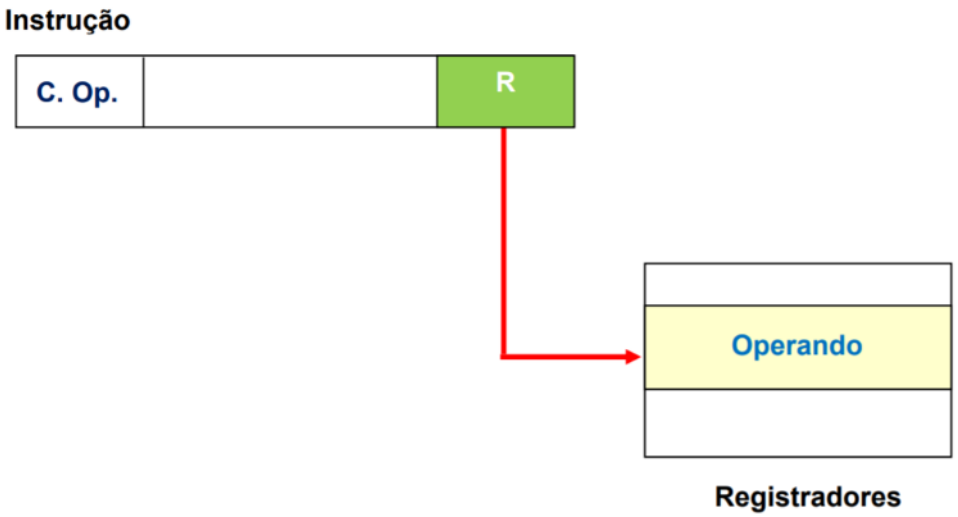
Figura 3 – Instrução do modo de endereçamento direto



Fonte: Alexandre Lucas Chichosz et al (8)

O modo de endereçamento por registrador possui o campo operando contendo uma referência a um registrador que contém o operando. No caso, o campo de registrador tem 5 *bits*, assim pode referenciar até 32 registradores diferentes. As suas vantagens são: possuir um campo pequeno de referência ao registrador e não usa referência de memória principal, conforme a Figura 4.

Figura 4 – Instrução do modo de endereçamento por registrador



Fonte: Alexandre Lucas Chichosz et al (8)

O modo de endereçamento por pilha é usado para referenciar a pilha presente na memória de dados. Um registrador padrão que indica o topo da pilha é referenciado implicitamente para realizar operações de empilhamento e desempilhamento na pilha.

3.5 Taxonomia de Flynn

A taxonomia de Flynn é uma técnica criada por Michael J. Flynn (10) que é usada até hoje para classificar de sistemas computacionais pelo seu tipo de processamento de dados. De acordo com William Stallings em *Computer Organization and Architecture* (7, p. 630, tradução nossa), esse modelo de taxonomia abrange quatro classes de arquitetura de computadores:

- SISD (*Single Instruction Single Data*): fluxo único de instruções sobre um único conjunto de dados.
- SIMD (*Single Instruction Multiple Data*): fluxo único de instruções em múltiplos conjuntos de dados.
- MISD (*Multiple Instruction Single Data*): fluxo múltiplo de instruções em um único conjunto de dados.
- MIMD (*Multiple Instruction Multiple Data*): fluxo múltiplo de instruções sobre múltiplos conjuntos de dados.

3.6 Arquitetura RISC

Os processadores podem ser subdivididos em duas linhas quando se trata do tipo de instrução que executam: CISC (*Complex Instruction Set Computer*) e RISC (*Reduced Instruction Set Computer*). Os processadores do tipo RISC, de acordo com William Stallings em *Computer Organization and Architecture* (7), busca otimizar a eficiência da execução de comandos priorizando instruções básicas como a movimentação de dados e atribuição de valores.

3.7 *Quartus Prime*

O *software* utilizado para a implementação e simulação neste trabalho será o *Intel Quartus Prime Design* da Intel (11). Ele foi projetado para FPGAs, SoCs e CPLDs Intel, desde a entrada e síntese do projeto até a otimização, verificação e simulação. Sua capacidade foi drasticamente aumentada comparado com as versões anteriores e possui vários elementos lógicos fornecendo aos projetistas a plataforma ideal para aproveitar as oportunidades de design da próxima geração. A versão *Lite* é gratuita e está disponível para *Linux* e *Windows*.

3.8 Linguagem de Descrição de *Hardware*

Uma alternativa à entrada esquemática de um circuito digital em um sistema de projeto auxiliado por computador é utilizar a técnica de projeto de dispositivos lógicos programáveis (PLDs) com uma ferramenta de projeto baseado em texto ou linguagem de descrição de *hardware* (HDL). Exemplos de HDLs são o AHDL - *Altera Hardware Description Language* , e os padrões VHDL e Verilog (12). A linguagem utilizada neste trabalho será o Verilog.

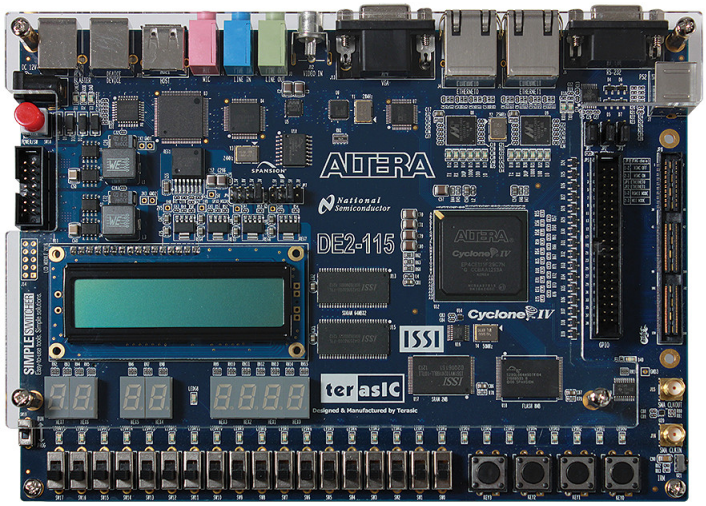
Uma das características de uma linguagem de descrição *hardware* é que ela é independente do *hardware*, ou seja, compatível para ser empregada em diferentes dispositivos.

3.9 FPGA - *Field-Programmable Gate Array*

FPGA, ou *Field-Programmable Gate Array*, é um circuito integrado reprogramável. Segundo Fabbrycio Cardoso da UNICAMP, "um chip FPGA contém um grande número de unidades lógicas que podem ser interconectadas através de uma matriz de trilhas condutoras e de *switches* programáveis". Cada unidade lógica de uma FPGA pode ser configurada de forma independente, o que torna possível modificar o circuito caso o *hardware* sofra alterações.

O kit FPGA utilizado para o desenvolvimento e simulação do projeto foi o *Altera DE2-115 Development and Education Board* mostrado na Figura 5, que contém o FPGA *Altera Cyclone IV E: EP4CE115F29C7* (13).

Figura 5 – FPGA

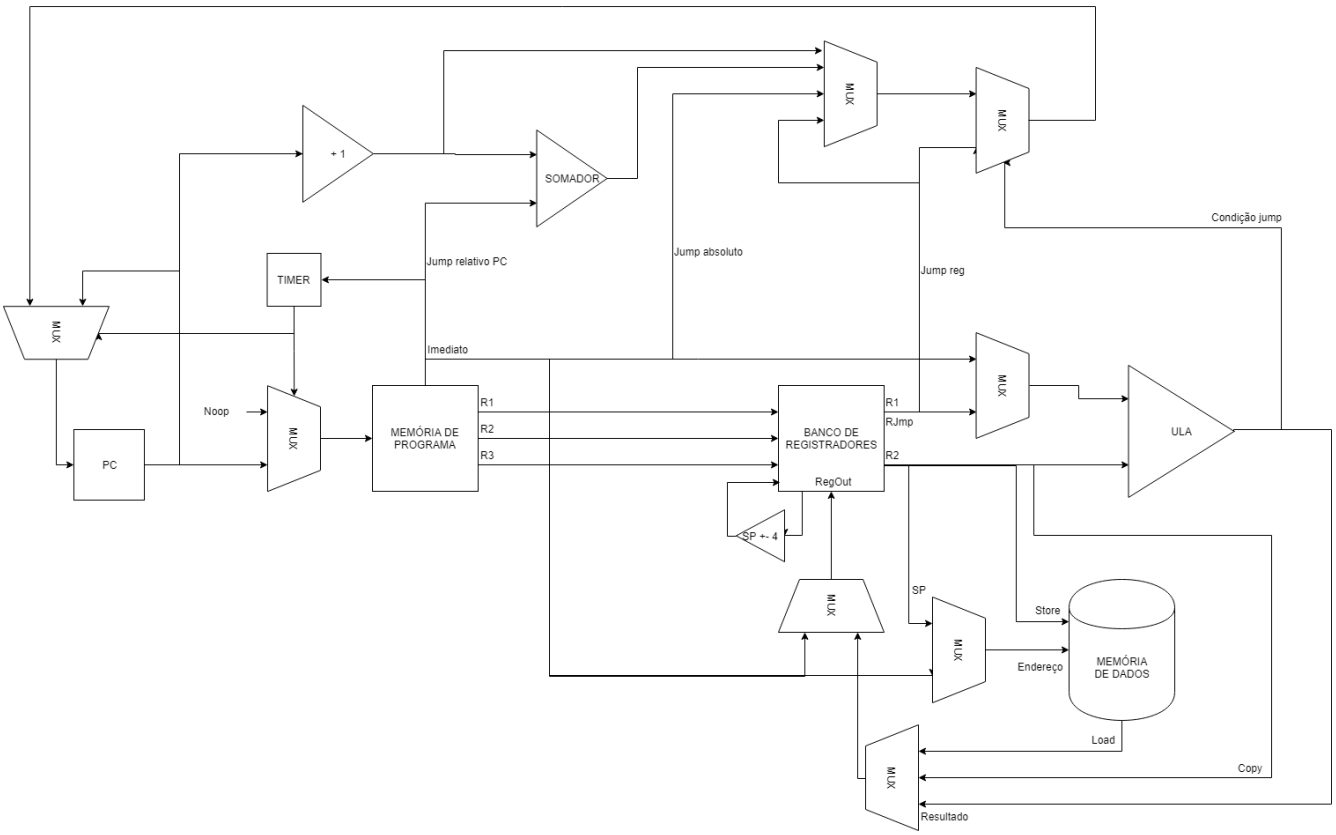


Fonte: *Altera DE2-115*. (13)

4 Desenvolvimento

A base para o desenvolvimento do projeto do processador foi o diagrama ilustrado na Figura 6. Nele, é possível identificar componentes essenciais do processador, como o registrador PC, as memórias de programa e de dados, a unidade lógica aritmética e o banco de registradores, onde vale ressaltar a presença de registradores especiais como o registrador SP, que armazena o endereço do topo da pilha presente na memória de dados, e os registradores MS/LS e Quo/Rem, que armazenam, respectivamente, os resultados das operações de multiplicação e divisão.

Figura 6 – Esquemático do processador

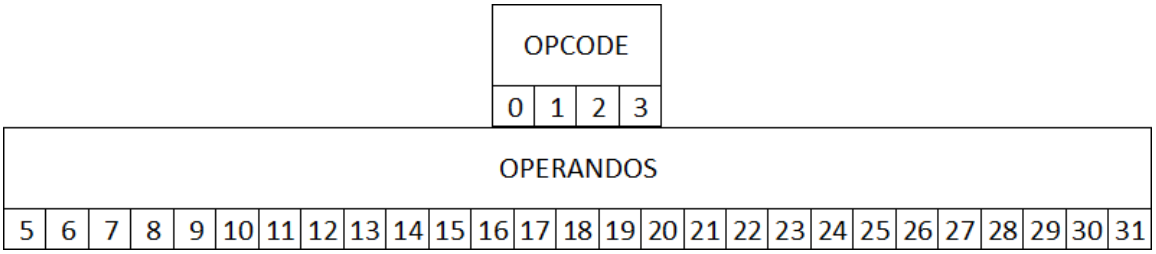


Fonte: O Autor

4.1 Conjunto de Instruções

As instruções possuem 32 *bits* de comprimento, sendo 4 *bits* reservados para a identificação da instrução (*OPCODE*), permitindo a existência de até 16 tipos de instrução diferentes, e 28 *bits* para operandos, como mostra a Figura 7.

Figura 7 – Formato da instrução



Fonte: O Autor

As instruções que poderão ser interpretadas pelo processador foram separadas em 14 grupos, cada grupo sendo identificado por um *OPCODE* diferente de acordo com a Tabela 1

Tabela 1 – Tipos de instrução

OPCODE	INSTRUÇÃO
0000	SOMA
0001	SUBTRAÇÃO
0010	MULTIPLICAÇÃO
0011	DIVISÃO
0100	DESLOCAMENTO
0101	LÓGICA
0110	*JUMP*
0111	PILHA
1000	*WRITE*
1001	*COPY*
1010	*LOAD*
1011	*STORE*
1100	*SLEEP*
1100	*INPUT*

Fonte: O Autor

A seguir, serão apresentados os formatos dos 28 *bits* de operandos para cada tipo de instrução.

4.1.1 Soma e Subtração

As instruções dos tipos **soma** e **subtração** foram divididas em dois subtipos diferenciados pela *flag* T. Quando T possui o valor 0, a instrução é do subtipo 1, Figura 8(a), e armazena no registrador (REG. OUT) o resultado da soma/subtração entre os conteúdos armazenados nos registradores (REG. IN1) e (REG. IN2) . Quando T possui o valor 1, a instrução é do subtipo 2, Figura 8(b), e armazena no registrador (REG. OUT) o resultado da soma/subtração entre o conteúdo armazenado no registrador (REG. IN1) e o imediado (IMED. IN2).

Figura 8 – Operandos das instruções **soma** e **subtração**

(a) Soma/subtração - subtipo 1

REG. OUT					REG. IN1					REG. IN2					NULL											T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

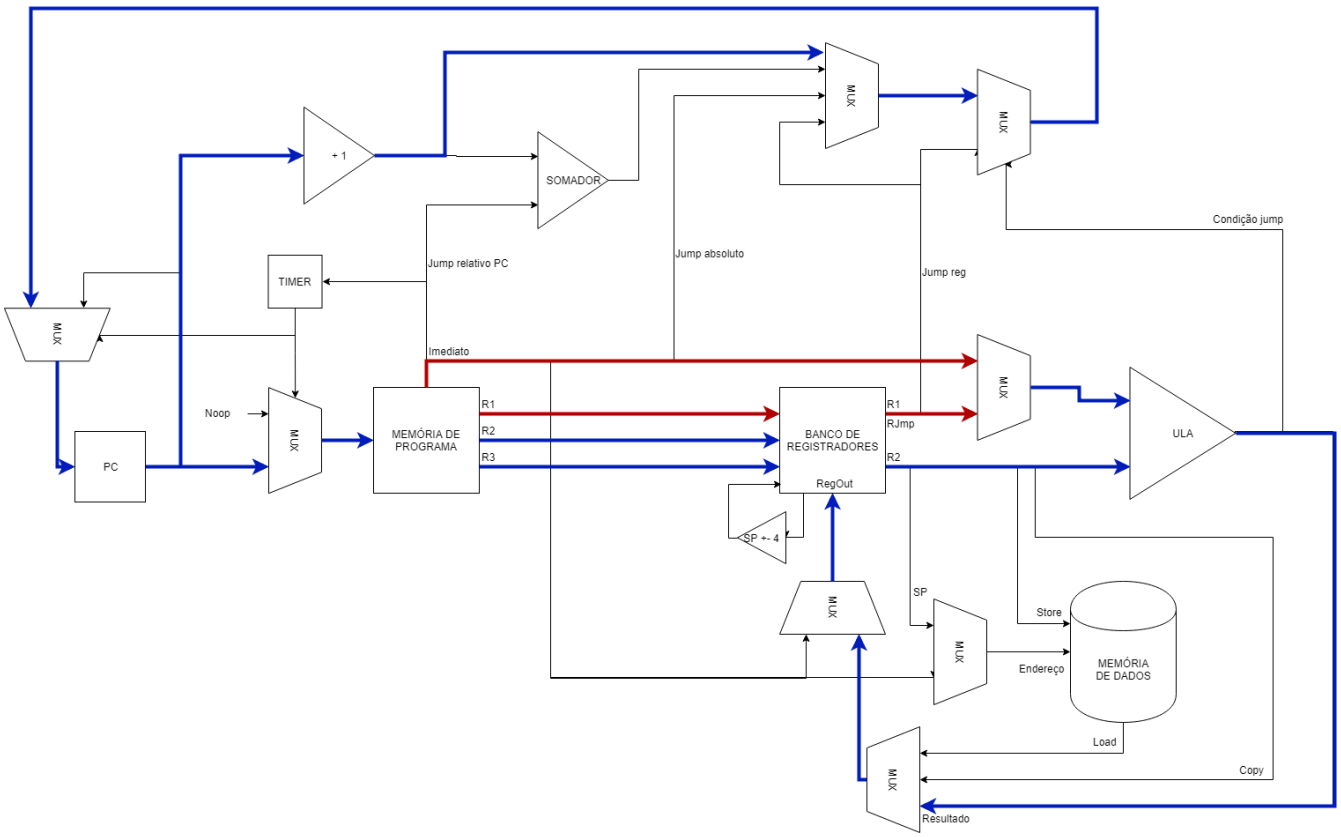
(b) Soma/subtração - subtipo 2

REG. OUT					REG. IN1					IMED. IN2																T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesses tipos de instrução são: **por registrador** para a saída (REG. OUT) e para as entradas (REG. IN1) e (REG. IN2), a segunda quando a *flag* assume o valor 0; e **imediato** para o segundo valor de entrada (IMED. IN2) quando a *flag* T assume o valor 1. O caminho de execução da instrução (*datapath*) pode ser observado na Figura 9, onde as setas em azul indicam caminhos tomados por todas as instruções desse tipo, enquanto as setas em vermelho indicam caminhos opcionais de acordo com a segunda entrada da operação.

Figura 9 – *Datapath* das instruções **soma** e **subtração**



Fonte: O Autor

4.1.2 Multiplicação e Divisão

As instruções dos tipos **multiplicação** e **divisão** também foram divididas em dois subtipos diferenciados pela *flag* T, que indica se a operação deve ser realizada entre os conteúdos de dois registradores, Figura 10(a), ou entre o conteúdo de um registrador e um valor imediato, Figura 10(b). A diferença entre esses tipos de insruções e as instruções de soma/subtração é que o resultado será armazenado em registradores específicos, reservados para aquele tipo de instrução; ou seja, os *bits* mais significativos do resultado de uma multiplicação são armazenados no registrador reservado **MS**, e os menos significativos no registrador **LS**. No caso da divisão, o quociente é armazenado no registrador **Quo** e o resto no registrador **Rem**.

Figura 10 – Operandos das instruções **multiplicação** e **divisão**

(a) Multiplicação/divisão - subtipo 1

REG. IN1					REG. IN2					NULL																T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

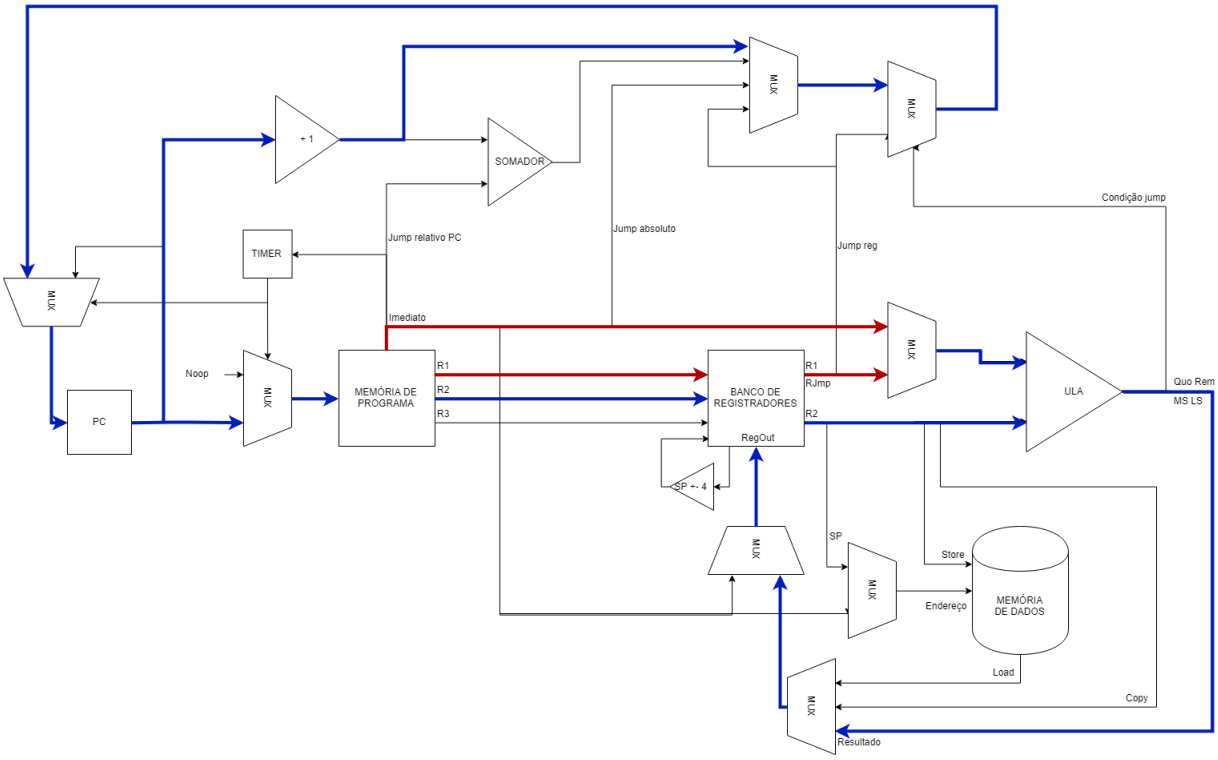
(b) Multiplicação/divisão - subtipo 2

REG. IN1					IMED. IN2																					T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesses tipos de instrução são: **por registrador** para as entradas (REG. IN1) e (REG. IN2), a segunda quando a *flag* assume o valor 0; e **imediato** para o segundo valor de entrada (IMED. IN2) quando a *flag* T assume o valor 1. O caminho de execução da instrução (*datapath*) pode ser observado na Figura 11, onde as setas em azul indicam caminhos tomados por todas as instruções desse tipo, enquanto as setas em vermelho indicam caminhos opcionais de acordo com a segunda entrada da operação.

Figura 11 – *Datapath* das instruções **multiplicação** e **divisão**



Fonte: O Autor

4.1.3 Deslocamento

Essa instrução, usando a nomenclatura da Figura 12, armazena no registrador (Reg. OUT) o conteúdo do registrador (REG. IN1) deslocado (IMED. IN2) *bits*. O deslocamento é para a direita quando a *flag* T assume o valor 0, e para a esquerda quando assumo o valor 1.

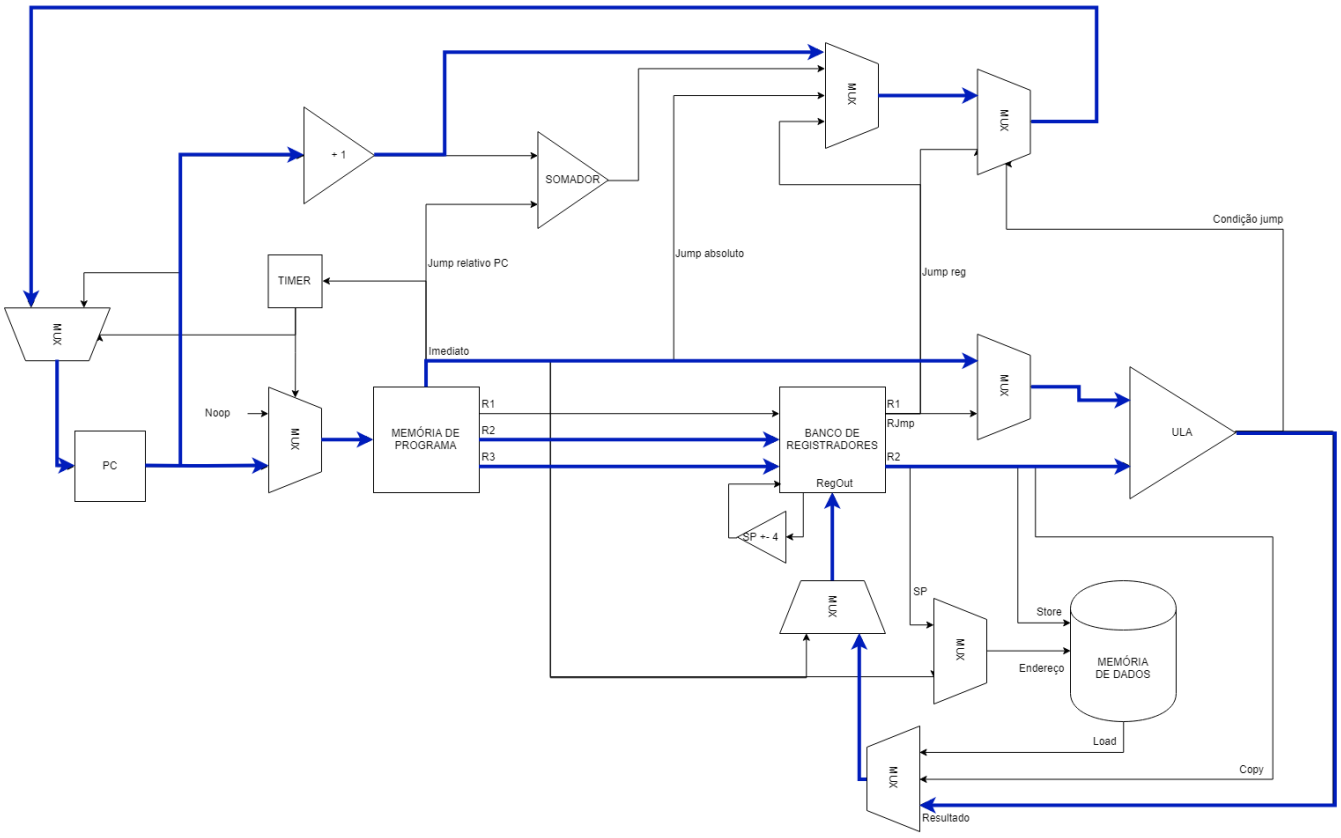
Figura 12 – Operandos das instruções de **deslocamento**

REG. OUT					REG. IN1					IME. IN2																T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a saída (REG. OUT) e para o primeiro valor de entrada (REG. IN), e **imediato** para o segundo valor de entrada (IMED. IN2). O caminho de execução da instrução, para ambos os sentidos de deslocamento, (*datapath*) pode ser observado na Figura 13.

Figura 13 – *Datapath* das instruções de **deslocamento**



Fonte: O Autor

4.1.4 Lógica

Esse tipo de instrução é dividido em 8 subtipos diferentes, identificados pela *flag* de 3 *bits* T de acordo com a Tabela 2. O resultado da operação lógica entre os registradores (REG. IN1) e (REG. IN2) é armazenado no registrador (REG. OUT), de acordo com a Figura 14(b), há uma exceção para a operação do tipo **NOT**, que utiliza apenas um registrador de entrada (REG. IN1), como mostra a Figura 14(a).

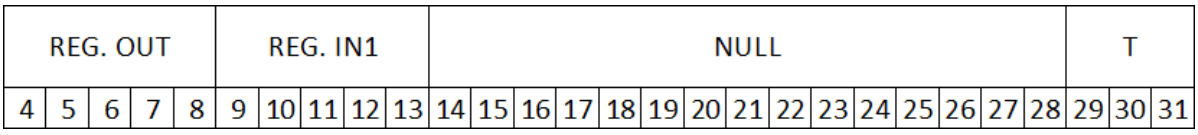
Tabela 2 – Subtipos de instrução lógica

T	OPERAÇÃO
000	*NOT*
001	*AND*
010	*OR*
011	*XOR*
100	IGUAL
101	DIFERENTE
110	MAIOR QUE
111	MENOR QUE

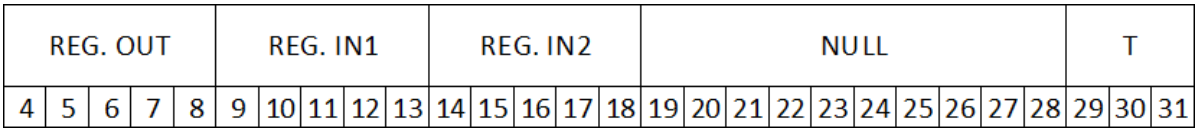
Fonte: O Autor

Figura 14 – Operandos das instruções do tipo **lógica**

(a) Lógica - tipo NOT



(b) Lógica - demais tipos

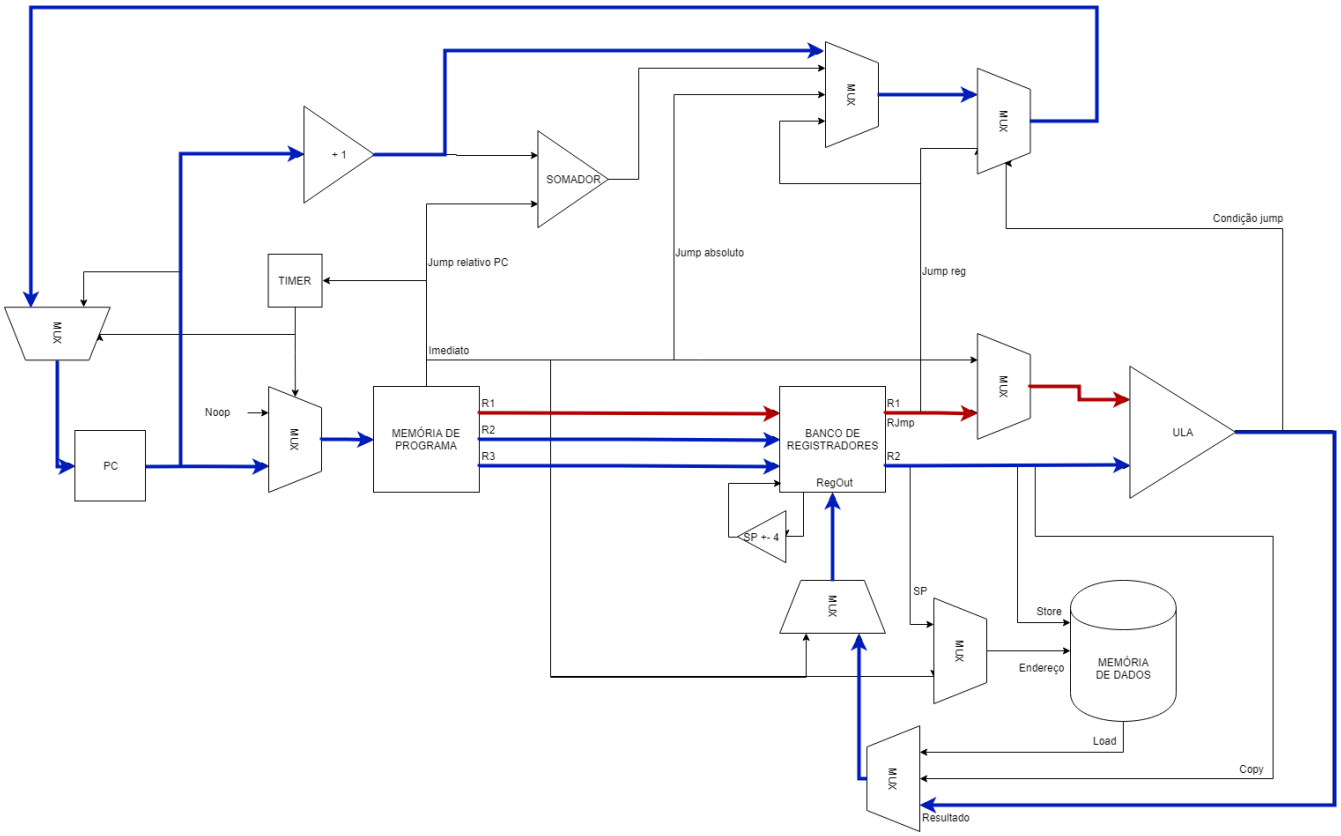


Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a saída (REG. OUT) e para o primeiro valor de entrada (REG. IN), além de para o segundo valor de entrada (REG. IN2) no caso de a operação não ser do subtipo NOT. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 15, onde as

setas em azul indicam o caminho tomado por qualquer subtipo de instrução lógica, e as setas em vermelho diferenciam a operação *NOT* das demais.

Figura 15 – *Datapath* das instruções do tipo **lógica**



Fonte: O Autor

4.1.5 *Jump*

As instruções desse tipo têm como objetivo desviar a contagem automática do registrador PC para acessar outras seções do programa. Essa operação é feita de quatro jeitos diferentes, identificados pela *flag* de 3 *bits* T de acordo com a Tabela 3. O primeiro tipo, "relativo ao PC por deslocamento", Figura 16(a), incrementa ou decrementa em (IMED. IN) o endereço armazenado no PC; ou seja, é relativo à sua posição atual. O segundo tipo, "absoluto direto", Figura 16(b), reposiciona o PC no endereço de programa (END. IN), independente de sua posição atual. O terceiro tipo, "absoluto por registrador", Figura 16(c), reposiciona o PC no endereço de programa armazenado no registrador (REG. JMP), independente de sua posição atual. O quarto tipo, "condicional absoluto por registrador", Figura 16(d), realiza a mesma operação que o terceiro tipo, porém somente sob a condição de que o conteúdo armazenado no registrador (REG. IN) represente o valor lógico **verdadeiro**.

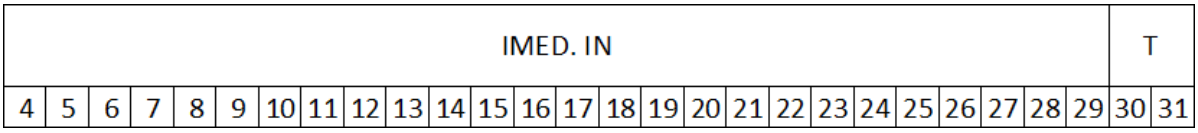
Tabela 3 – Subtipos de instrução *jump*

T	OPERAÇÃO
00	*Jump* relativo ao PC por deslocamento
01	*Jump* absoluto direto
10	Jump absoluto por registrador
11	*Jump* condicional absoluto por registrador

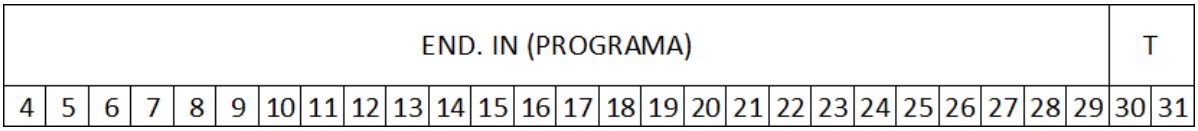
Fonte: O Autor

Figura 16 – Operandos das instruções do tipo ***jump***

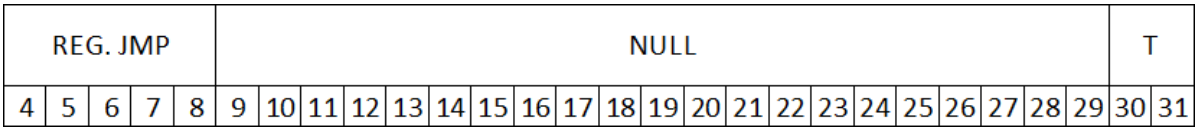
(a) Jump - relativo ao PC por deslocamento



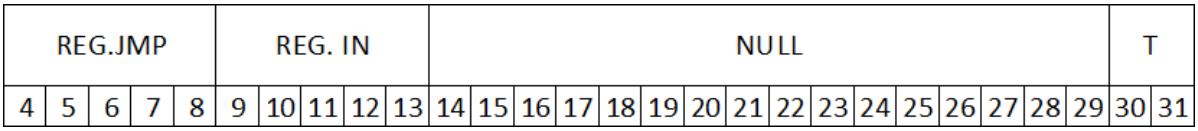
(b) *Jump* - absoluto direto



(c) *Jump* - absoluto por registrador



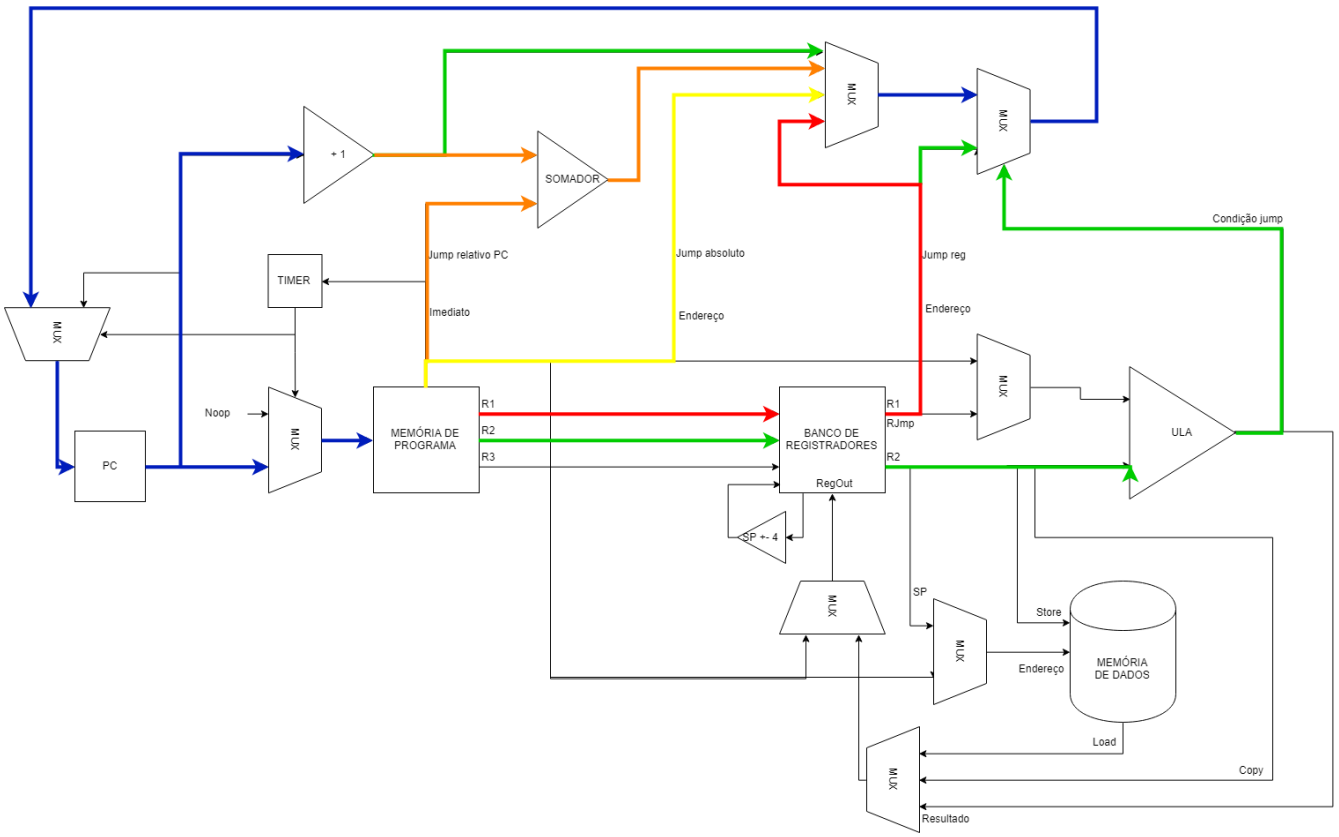
(d) *Jump* - condicional absoluto por registrador



Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para as entradas (REG. JMP) e (REG. IN), nos subtipos aos quais fazem parte, **imediado** para a entrada (IMED. IN) do primeiro subtipo, e **direto** para a entrada (END. IN) do segundo subtipo. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 17, onde as setas em azul indicam o caminho tomado por qualquer subtipo de instrução, as setas em laranja indicam o caminho tomado pelo primeiro subtipo, a em amarelo o caminho tomado pelo segundo subtipo, as em vermelho para o terceiro e as setas em verde indicam opções de caminhos tomados pelo quarto subtipo de instrução *jump*.

Figura 17 – *Datapath* das instruções do tipo ***jump***



Fonte: O Autor

4.1.6 Pilha

Esse tipo de instrução, usando a nomenclatura da Figura 18, realiza as operações de **empilhamento**, quando a *flag* T assume o valor 0, do conteúdo de um registrador (REG. IO) na pilha presente na memória principal; e **desempilhamento**, quando T assume o valor 1, do conteúdo presente no topo da pilha, armazenando-o no registrador (REG. IO).

Figura 18 – Operandos das instruções do tipo **pilha**

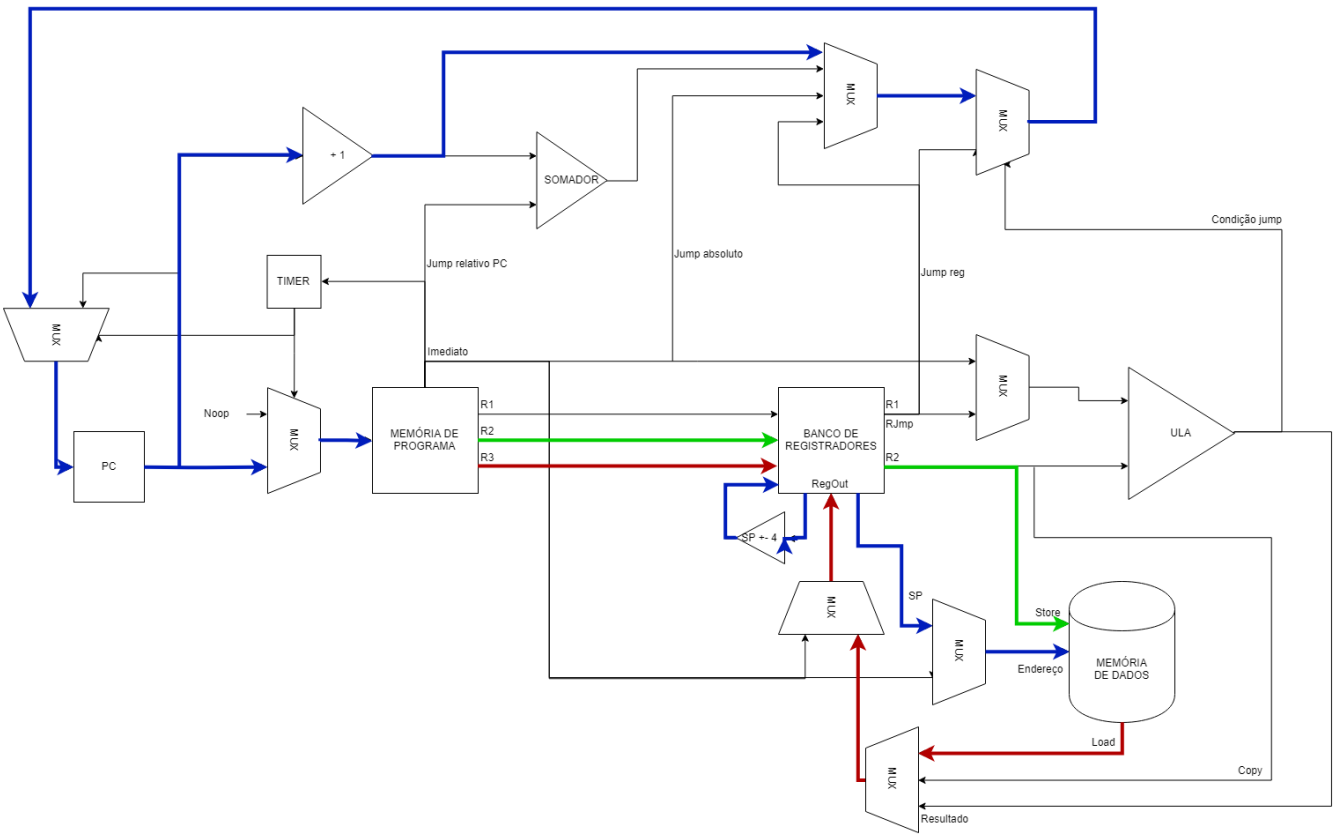
REG. IO					NULL																					T
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a entrada/saída (REG. IO) e **por pilha** para referenciar o topo da pilha (SP) na memória de dados. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 19, onde as setas em azul indicam o caminho tomado tanto por empilhamento quando por desempilhamento de dados, as setas em verde indicam o caminho tomado

exclusivamente pelo empilhamento, e as setas em vermelho indicam o caminho tomado exclusivamente pelo desempilhamento.

Figura 19 – *Datapath* das instruções do tipo **pilha**



Fonte: O Autor

4.1.7 Write

Essa instrução armazena o valor (IMED. IN) no registrador (REG. OUT), Figura 20.

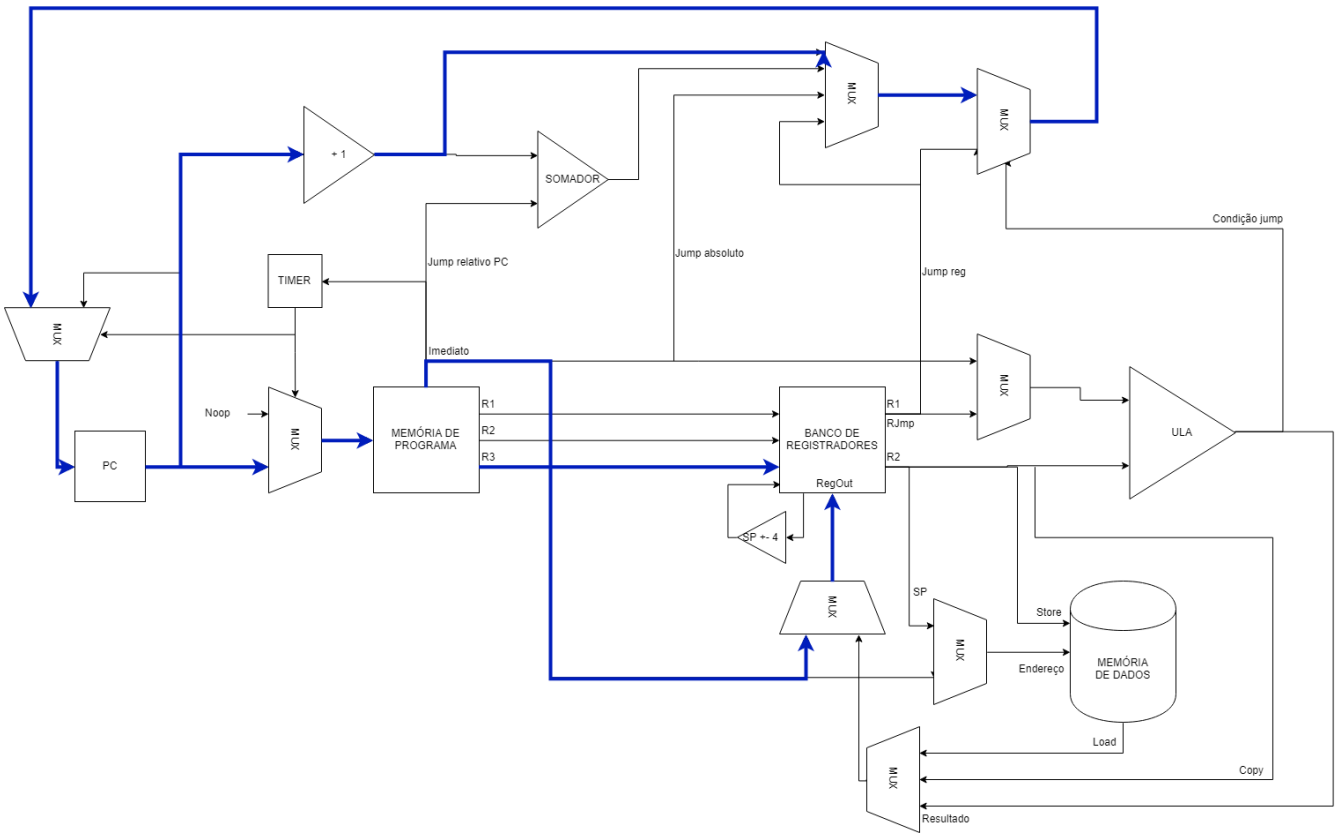
Figura 20 – Operandos da instrução ***write***

REG. OUT					IME. IN																					
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para a saída (REG. OUT) e **imediato** para a entrada (IMED. IN) a ser armazenada no registrador. O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 21.

Figura 21 – *Datapath* da instrução ***write***



Fonte: O Autor

4.1.8 Copy

Essa instrução copia o conteúdo do registrador (REG. IN) no registrador (REG. OUT), Figura 22. Essa instrução também é utilizada para alterar o valor da saída, copiando o valor de um registrador para o registrador de saída.

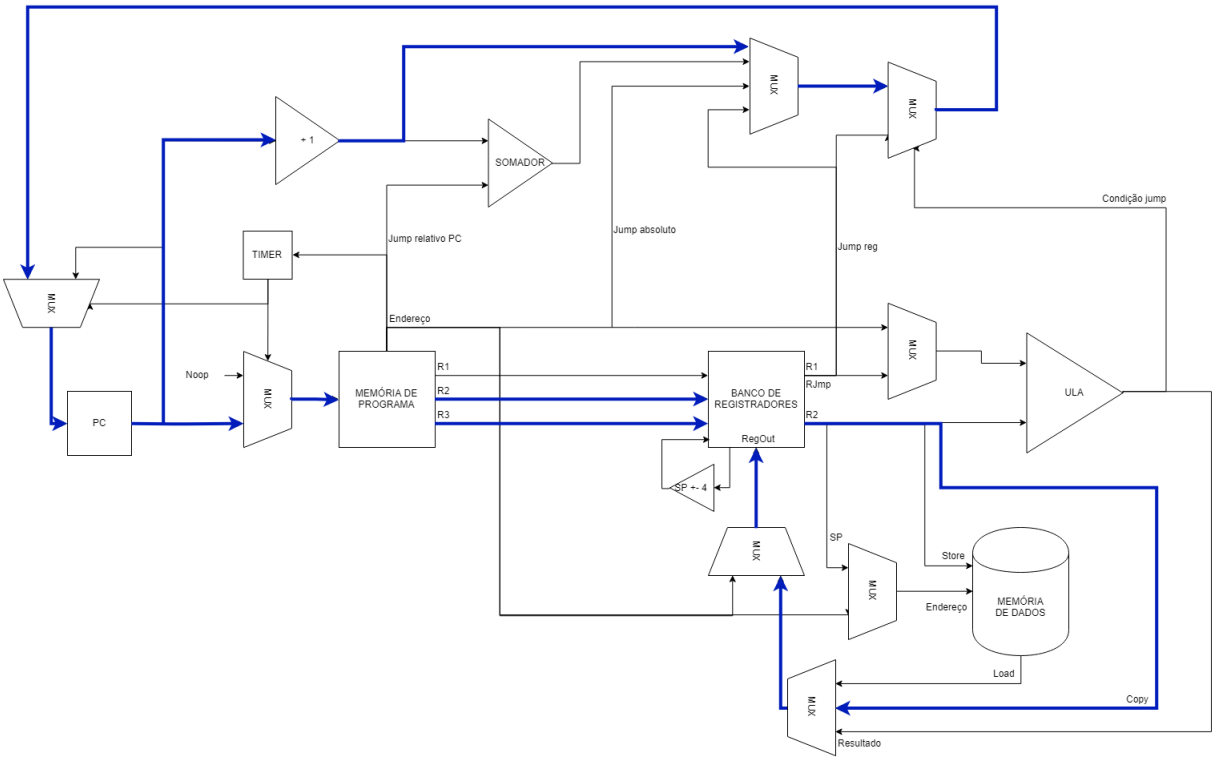
Figura 22 – Operandos da instrução ***copy***

REG. OUT					REG. IN					NULL																
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

O modo de endereçamento utilizado nesse tipo de instrução é: **por registrador**, tanto para para a entrada (REG. IN) quanto para a saída (REG. OUT). O caminho de execução da instrução, (*datapath*) pode ser observado na Figura 23.

Figura 23 – *Datapath* da instrução ***copy***



Fonte: O Autor

4.1.9 Load e Store

Esses tipos de instrução realizam movimentações na memória de dados. Load, Figura 24(a), carrega no registrador (REG. OUT) o conteúdo presente no endereço (END. IN) da memória de dados, enquanto Store, Figura 24(b), armazena o conteúdo do registrador (REG. IN) no endereço (END. OUT) da memória da dados.

Figura 24 – Operandos das instruções ***load*** e ***store***

(a) Load

REG. OUT					END. IN (DADOS)																			
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

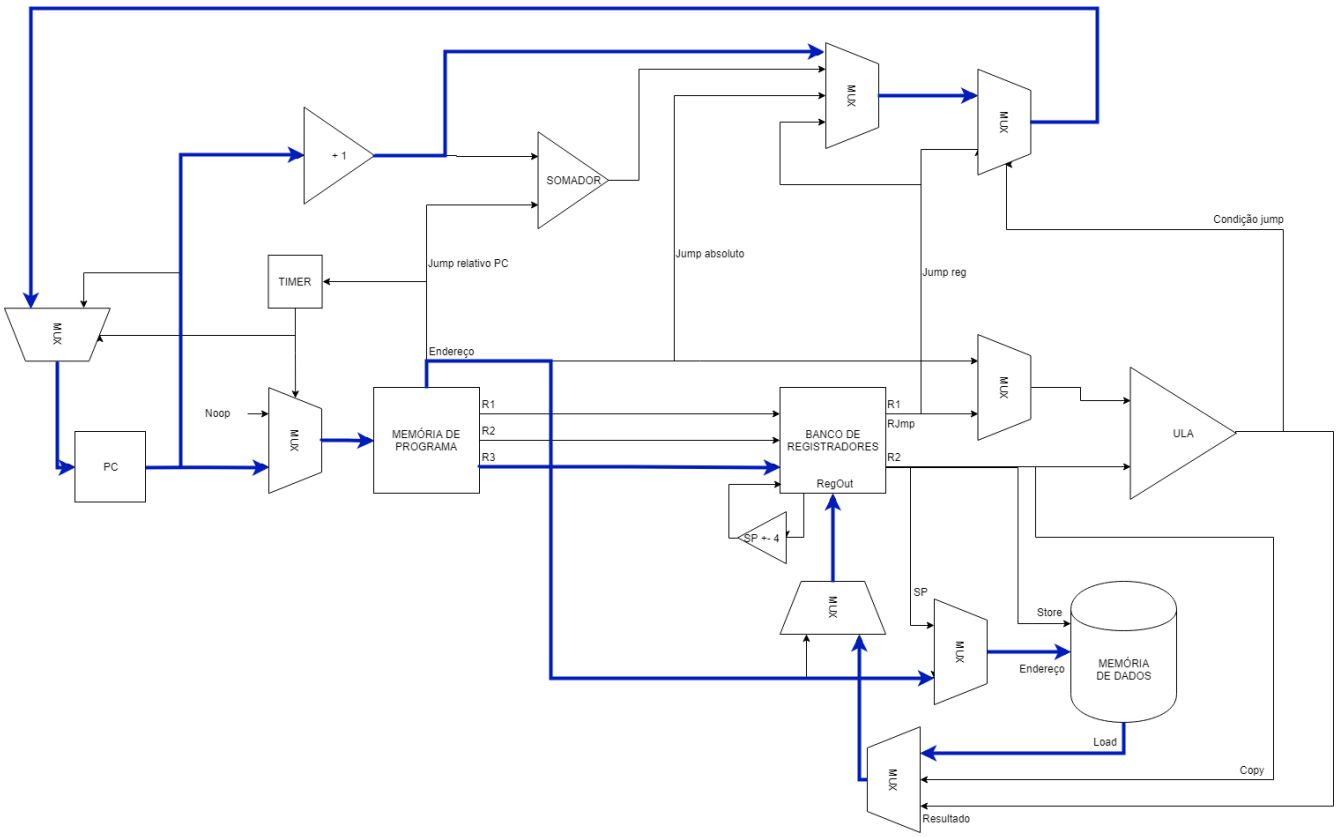
(b) Store

REG. IN					END. OUT (DADOS)																			
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Fonte: O Autor

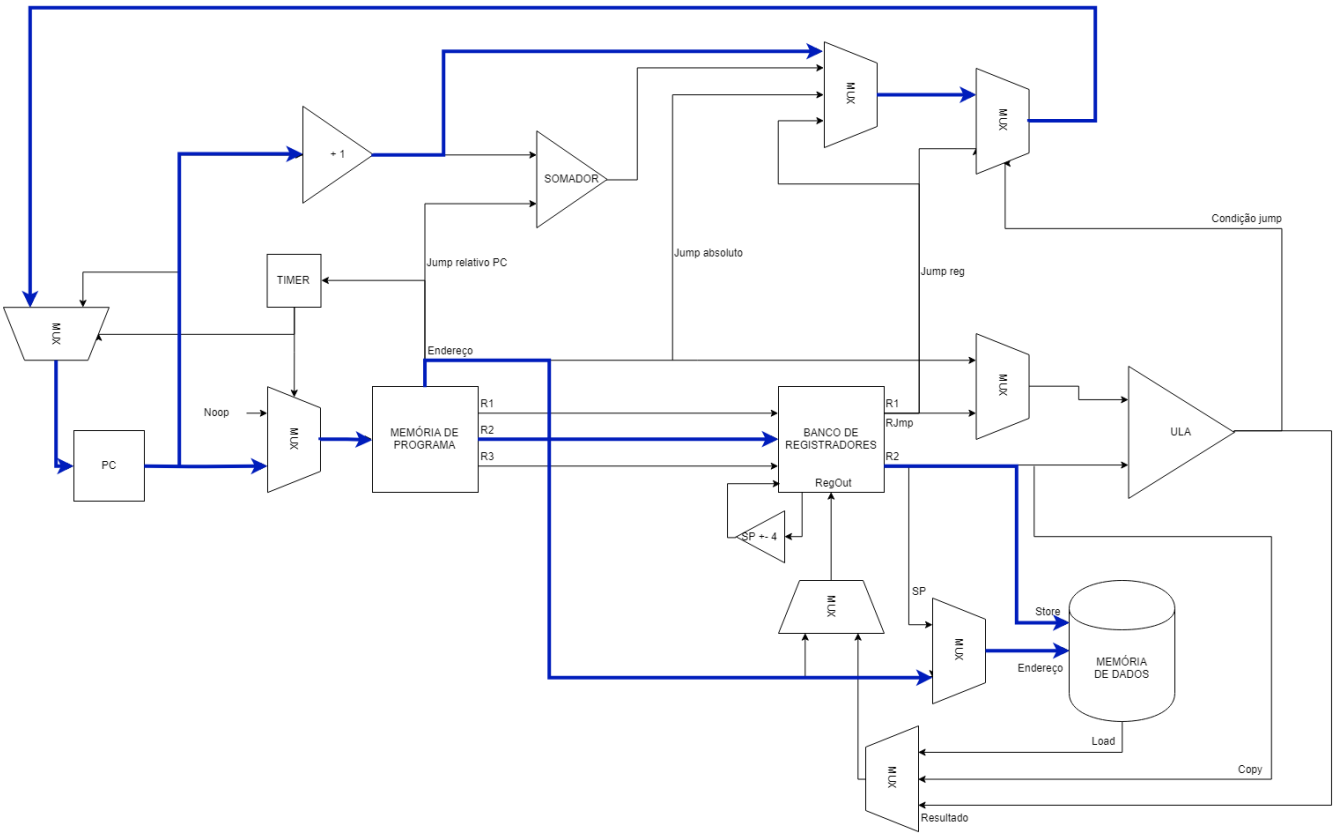
Os modos de endereçamento utilizados nesse tipo de instrução são: **por registrador** para o registrador de saída (REG. OUT), no caso da instrução store, e para o registrador de entrada (REG. IN), no caso da instrução load; e **direto** para a entrada (END. IN), no caso da instrução store, e para a saída (END. OUT), no caso da instrução load. O caminho de execução da instrução do tipo load pode ser observado na Figura 25, enquanto o da instrução do tipo store pode ser observado na Figura 26.

Figura 25 – *Datapath* da instrução ***load***



Fonte: O Autor

Figura 26 – *Datapath* da instrução ***store***



Fonte: O Autor

4.1.10 Sleep

A instrução sleep faz com que o processador pare de receber instruções (receba a instrução NOOP) e o PC pare de se movimentar pelo tempo determinado, em milissegundos, por (TIME), Figura 27.

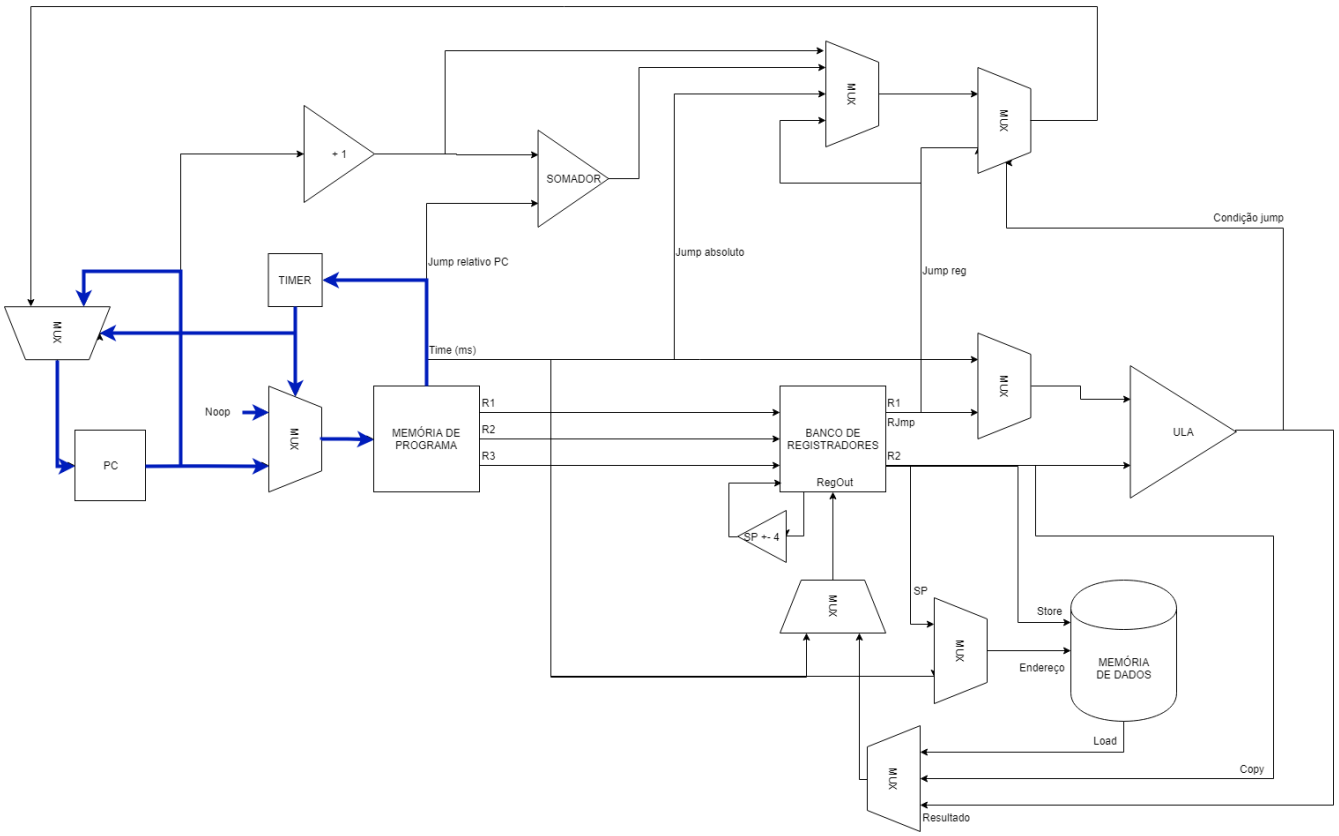
Figura 27 – Operandos da instrução ***sleep***

TIME (ms)																										
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Fonte: O Autor

O modo de endereçamento utilizado nesse tipo de instrução é: **imediado** para o tempo determinado na entrada (TIME). O caminho de execução da instrução pode ser observado na Figura 28.

Figura 28 – *Datapath* da instrução ***sleep***



Fonte: O Autor

4.1.11 Input

A instrução *input* faz com que o PC pare de se movimentar até que a entrada seja fornecida. Para confirmar a inserção da entrada, o usuário deve alternar duas vezes o estado do switch de entrada (ativar e desativar). O formato dessa instrução pode ser visualizado na Figura 29.

Figura 29 – Operandos da instrução ***input***

REG. OUT					INPUT																				
27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

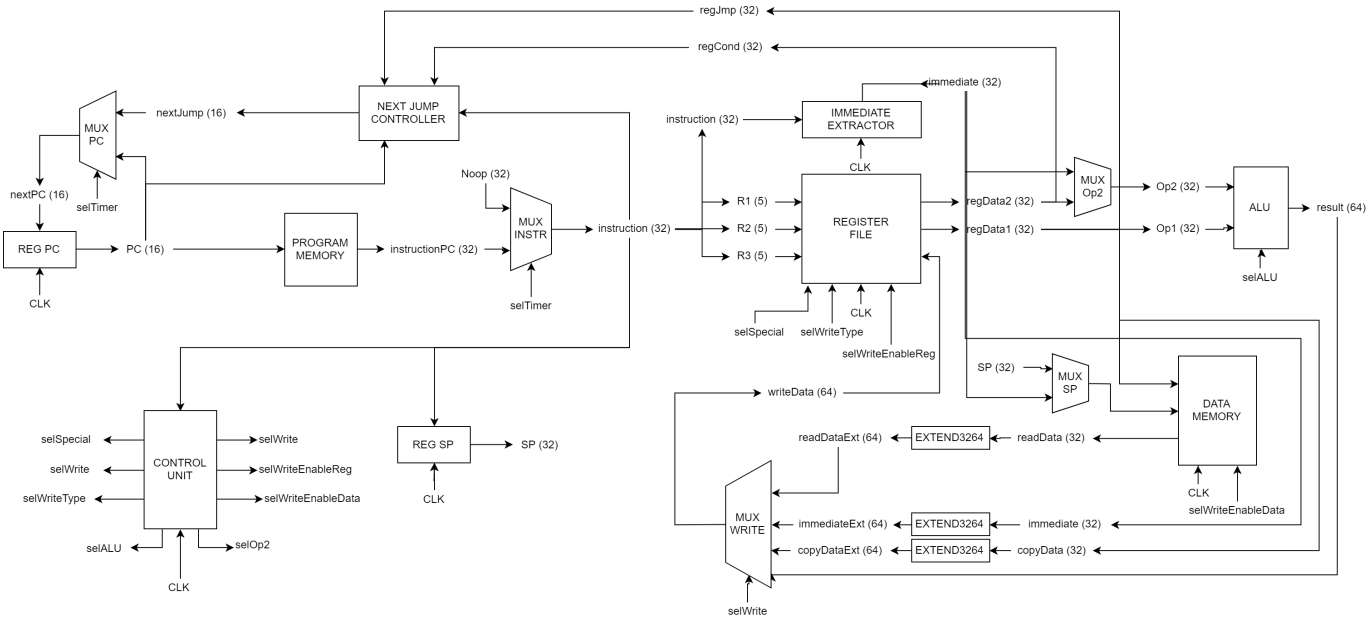
Fonte: O Autor

O único modo de endereçamento utilizado nesse tipo de instrução é por registrador para o indicar o registrador que receberá o valor de entrada.

4.2 Implementação dos blocos da unidade de processamento

Na segunda etapa do projeto, foram desenvolvidos os principais blocos que fazem parte da unidade de processamento do sistema computacional, incluindo a unidade lógica aritmética, o banco de registradores, as memórias de dados e de programa, além de diversos blocos intermediários que auxiliam na interligação entre eles e na manipulação das instruções. Para auxiliar na interpretação e na implementação da arquitetura do sistema, foi desenvolvido um novo esquemático, ilustrado na Figura 30 mais detalhado voltado para o entendimento da movimentação dos dados em etapa do processo. Em seguida, serão analisados com mais detalhes cada bloco individualmente, juntamente com os respectivos códigos implementados seguindo o memso padrão de nomes do esquemático detalhado.

Figura 30 – Novo esquemático detalhado.



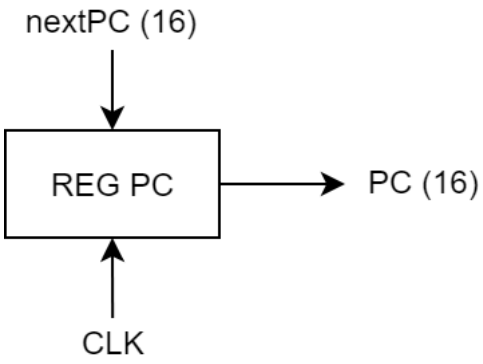
Fonte: O Autor

4.2.1 REG PC

O fio PC (Program Counter) de 16 bits serve como um indicador de qual instrução armazenada na memória de programa (PROGRAM MEMORY) deve ser executada no momento. O fio nextPC, também de 16 bits, indica qual será o valor de PC depois da próxima atualização.

O papel do bloco REG PC, ampliado na Figura 31 é simplesmente atualizar o valor do fio PC a cada descida de *clock* com o valor vigente de nextPC, que será especificado nas próximas seções. A implementação deste bloco pode ser visualizada no código REG PC.

Figura 31 – Bloco REG PC



Fonte: O Autor

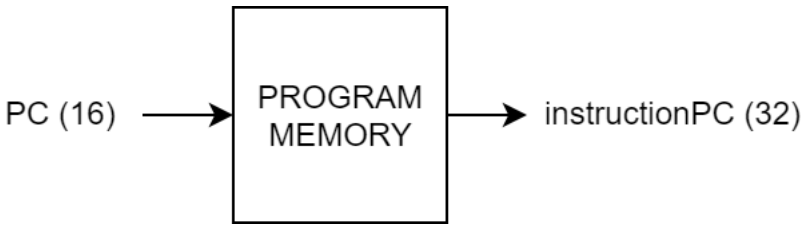
```
1  module regPC
2  #(parameter TAM_PC = 16)
3  (
4          input wire  [(TAM_PC-1):0] nextPC,
5          output wire [(TAM_PC-1):0] PC,
6          input wire  clk
7  );
8
9  reg [(TAM_PC-1):0] aux;
10 assign PC = aux;
11
12 always @ (negedge clk) aux = nextPC;
13
14 endmodule
```

4.2.2 PROGRAM MEMORY

É neste bloco, amplicado na Figura 32, que todas as instruções do programa que a ser executado ficam armazenadas. Além de armazenar as instruções, é função deste bloco atualizar o valor do fio instructionPC com o valor da instrução referente ao PC sempre que este for atualizado, ou seja, não depende diretamente do *clock*.

A implementação deste bloco, código PROGRAM MEMORY, mostra como a atualização de valor de instructionPC é realizada com base na atualização do fio PC. Para simulação, foi utilizada uma memória que comporta até 10 instruções, que são carregadas do arquivo "program.txt".

Figura 32 – Bloco PROGRAM MEMORY



Fonte: O Autor

```
1  module programMemory
2  #(parameter TAM_PC = 16)
3  (
4          input wire  [(TAM_PC-1):0] PC,
5          output wire [31:0] instructionPC
6  );
7
8  //reg [31:0] instructions[(2**TAM_PC)-1:0];
9  reg [31:0] instructions[9:0];
10
11 reg [31:0] regInstr;
12 assign instructionPC = regInstr;
13
14 initial begin
15         $readmemb("program.txt", instructions);
16 end
17
18 always @ (PC) regInstr = instructions[PC];
19
20 endmodule
```

4.2.3 REGISTER FILE

Este bloco, Figura 33, armazena informações nos 32 registradores de propósito geral e em alguns registradores especiais que guardam, por exemplo, o resultado de uma divisão ou multiplicação. Sendo assim, ele possui duas funções principais durante o ciclo de *clock*, uma durante a subida e outra durante a descida.

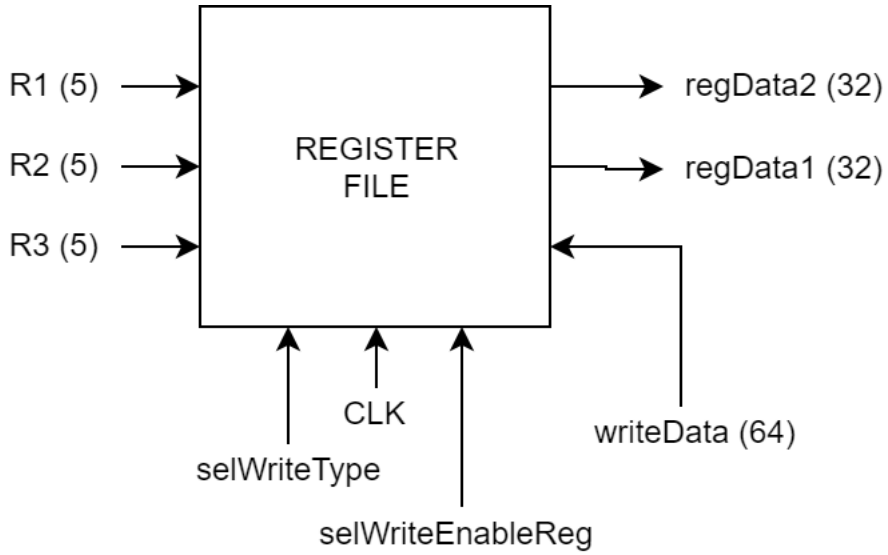
Na subida de clock, os valores de saída regData1 e regData2 são atualizados com base nos valores de entrada, respectivamente, R1 e R2, que indicam os índices dos registradores devem ter suas informações armazenadas expostas.

Na descida de *clock* é realizado, caso habilitado, o armazenamento do valor dos 32 bits menos significativos do fio writeData no registrador com índice especificado por R3. writeData possui 64 bits para que seja possível tratar exceções como os resultados de uma divisão ou multiplicação, que precisam ser armazenados em dois registradores de 32 bits cada.

A implementação do bloco representada pelo código REGISTER FILE mostra a separação das duas funcionalidades dele, a leitura de informações na subida de *clock* e a

escrita de informações, além do tratamento dos casos especiais de escrita usando o seletor selWriteType, na descida de *clock*.

Figura 33 – Bloco REGISTER FILE



Fonte: O Autor

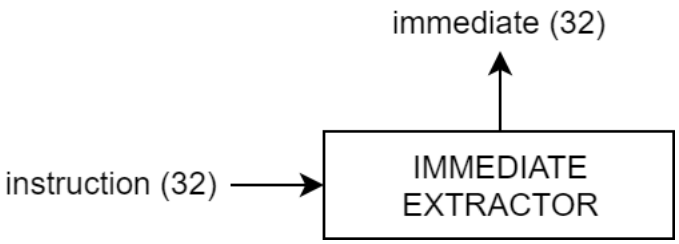
```
1  module regFile
2  (
3          input wire  [4:0]  R1,
4          input wire  [4:0]  R2,
5          input wire  [4:0]  R3,
6          output wire [31:0] regData1,
7          output wire [31:0] regData2,
8          input wire  [63:0] writeData,
9          input wire  [1:0]  selWriteType,
10         input wire                      selWriteEnableReg, clk
11 );
12
13 parameter REGULAR = 2'b00;
14 parameter MULT = 2'b01;
15 parameter DIV = 2'b10;
16
17 parameter MS = 0;
18 parameter LS = 1;
19 parameter QUO = 2;
20 parameter REM = 3;
21
22 reg [31:0] registers[31:0];
23 reg [31:0] special[3:0];
24
25 reg [31:0] aux1, aux2;
26 assign regData1 = aux1;
27 assign regData2 = aux2;
28
29 always @ (posedge clk) begin
30         aux1 <= registers[R1];
31         aux2 <= registers[R2];
32 end
```

```
33
34 always @ (negedge clk) begin
35         if (selWriteEnableReg) begin
36                 case (selWriteType)
37                         REGULAR: registers[R3] <= writeData[31:0];
38                         MULT: begin
39                                 special[MS] = writeData[63:32];
40                                 special[LS] = writeData[31:0];
41                         end
42                         DIV: begin
43                                 special[QUO] = writeData[63:32];
44                                 special[REM] = writeData[31:0];
45                         end
46                         default: registers[R3] <= writeData[31:0];
47                 endcase
48         end
49 end
50
51 endmodule
```

4.2.4 IMMEDIATE EXTRACTOR

Na Figura 34 foi ampliado o bloco IMMEDIATE EXTRACTOR, seu papel é extrair das instruções pertinentes vindas do fio instruction valores que não correspondem à endereços de registradores, ou seja, operandos imediatos que serão utilizados em operações aritméticas, corresponder à endereços da memória de dados, entre outros. Além disso, o bloco identifica o tamanho do imediato com base na instrução recebida e extende seu comprimento para um padrão de 32 *bits*. O código implementado, IMMEDIATE EXTRACTOR, mostra como é feita a identificação do tipo de instrução pelo *OPCODE* e como é tratada a extração do imediato em cada caso.

Figura 34 – Bloco IMMEDIATE EXTRACTOR



Fonte: O Autor

```
1 module immediateExtractor
2 (
3         input wire  [31:0] instruction,
4         output wire [31:0] immediate,
5         input wire clk
6 );
7
8 parameter SUM                   = 4'b0000;
9 parameter SUBTRACT      = 4'b0001;
```

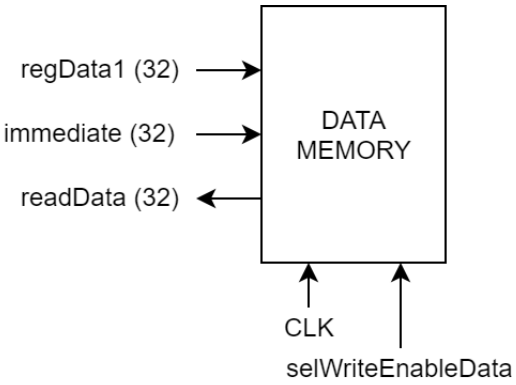
```
10 parameter MULTIPLY      = 4'b0010;
11 parameter DIVIDE                = 4'b0011;
12 parameter SHIFT         = 4'b0100;
13 parameter LOGIC         = 4'b0101;
14 parameter JUMP                  = 4'b0110;
15 parameter STACK         = 4'b0111;
16 parameter WRITE         = 4'b1000;
17 parameter COPY                  = 4'b1001;
18 parameter LOAD                  = 4'b1010;
19 parameter STORE         = 4'b1011;
20 parameter SLEEP         = 4'b1100;
21
22 reg [3:0] opcode;
23
24 reg  [31:0] auxImmediate;
25 assign immediate = auxImmediate;
26
27 always @ (posedge clk) begin
28         opcode = instruction[31:28];
29
30         case (opcode)
31                 SUM, SUBTRACT:          auxImmediate = instruction[0:0] == 1'b0 ? 32'b0 :
                        {15'b0, instruction[17:1]};
32                 MULTIPLY, DIVIDE: auxImmediate = instruction[0:0] == 1'b0 ? 32'b0 : {10'
                    b0, instruction[22:1]};
33                 SHIFT:                          auxImmediate = {10'b0, instruction
                    [17:1]};
34                 LOGIC:                          auxImmediate = {10'b0, instruction
                    [17:1]};
35                 JUMP:                                   auxImmediate = instruction[1:0]
                    == 2'b00 || instruction[1:0] == 2'b01 ? {6'b0, instruction[27:2]} :
                    32'b0;
36                 WRITE:                          auxImmediate = {9'b0, instruction[22:0]};
37                 LOAD, STORE:            auxImmediate = {10'b0, instruction[22:1]};
38                 SLEEP:                          auxImmediate = {4'b0, instruction[27:0]};
39                 default:                                auxImmediate = 32'b0;
40         endcase
41 end
42
43 endmodule
```

4.2.5 DATA MEMORY

A Figura 35 corresponde à ampliação do bloco DATA MEMORY, que armazena uma quantidade maior de informações que o bloco REGISTER FILE e que não passarão por movimentações com tanta frequência Seu funcionamento é parecido com o do bloco REGISTER FILE, passando por processos de leitura ou escrita dependendo do momento do ciclo de *clock*, subida ou descida. Diferente do bloco REGISTER FILE, este bloco não possui casos especiais de escrita, armazenando sempre dados de 32 *bits* e dependendo apenas da habilitação do seletor selWriteEnableData. Além disso, a definição do endereço de leitura ou escrita é feita pelo imediato de 32 *bits*, podendo abranger uma quantidade muito superior de endereços diferentes. O código DATA MEMORY se refere à implementação desse bloco e mostra a utilização de uma memória de capacidade muito superior ao

REGISTER FILE, além de maior simplicidade no processo de escrita.

Figura 35 – Bloco DATA MEMORY



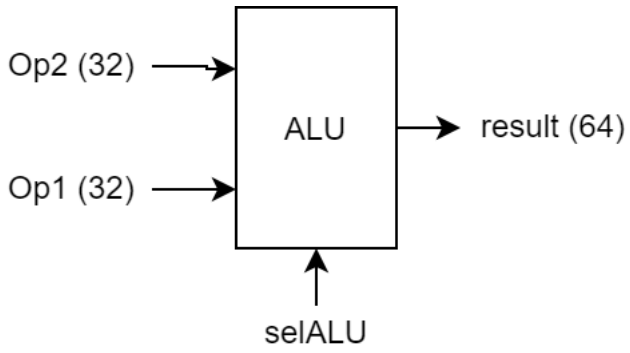
Fonte: O Autor

```
1  module dataMemory
2  (
3          input wire  [31:0] immediate,
4          input wire  [31:0] regData1,
5          output wire [31:0] readData,
6          input wire                      selWriteEnableData, clk
7  );
8
9  reg [31:0] data[(2**19):0]; // ~1MB
10
11 reg [31:0] aux1, aux2;
12 assign readData = aux2;
13
14 always @ (posedge clk) begin
15         aux1 <= data[immediate];
16 end
17
18 always @ (negedge clk) begin
19         if (selWriteEnableData)
20                 data[immediate] <= regData1;
21 end
22
23 endmodule
```

4.2.6 ALU

O bloco ALU foi ampliado na Figura 36, onde é possível observar que ele possui duas entradas de 32 *bits*, Op1 e Op2, além do seletor selALU, e uma saída de 64 *bits*, result. Este bloco realiza a operação aritmética selecionada por selALU entre os operandos vindos de Op1 e Op2, gerando um resultado de até 64 bits (variando de tamanho dependendo do tipo de operação) em result. O código ALU mostra todos os tipos de operação selecionáveis e o que cada um representa.

Figura 36 – Bloco ALU



Fonte: O Autor

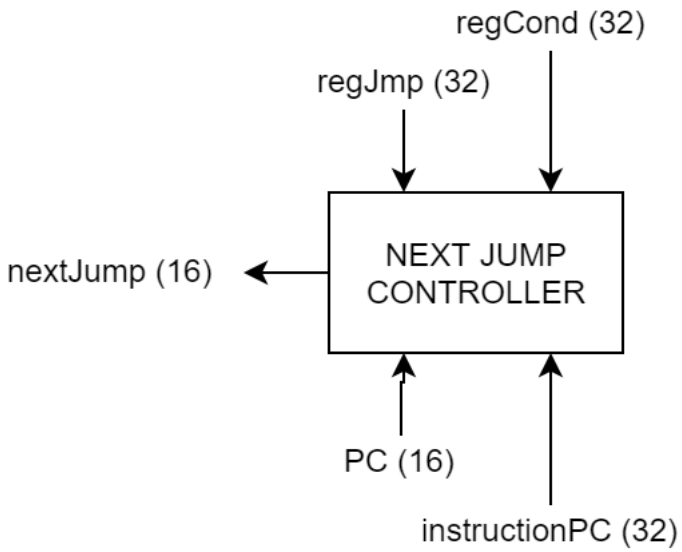
```
1  module alu (
2          input  [31:0] op1,
3          input  [31:0] op2,
4          output [63:0] result,
5          input  [3:0]  selALU
6  );
7
8  parameter SUM                = 4'b0000;
9  parameter SUBTRACT           = 4'b0001;
10 parameter MULTIPLY           = 4'b0010;
11 parameter DIVIDE             = 4'b0011;
12 parameter SHIFT_LEFT         = 4'b0100;
13 parameter SHIFT_RIGHT        = 4'b0101;
14 parameter BITWISE_NOT        = 4'b0110;
15 parameter BITWISE_AND        = 4'b0111;
16 parameter BITWISE_OR         = 4'b1000;
17 parameter BITWISE_XOR        = 4'b1001;
18 parameter LOGIC_EQUAL        = 4'b1010;
19 parameter LOGIC_DIFFERENT    = 4'b1011;
20 parameter LOGIC_LESS_THAN    = 4'b1100;
21 parameter LOGIC_GREATER_THAN = 4'b1101;
22
23 reg [31:0] aux1, aux2;
24
25 assign result = {aux2, aux1};
26
27 always @(op1 or op2) begin
28         aux2 = 32'b0;
29         case (selALU)
30                 SUM:                aux1 = op1 + op2;
31                 SUBTRACT:           aux1 = op1 - op2;
32                 MULTIPLY:           {aux2, aux1} = op1 * op2;
33                 DIVIDE: begin
34                                     aux2 = op1 / op2;
35                                     aux1 = op1 % op2;
36                 end
37                 SHIFT_LEFT:         aux1 = op1 << op2;
38                 SHIFT_RIGHT:        aux1 = op1 >> op2;
39                 BITWISE_NOT:        aux1 = ~op1;
40                 BITWISE_AND:        aux1 = op1 & op2;
41                 BITWISE_OR:         aux1 = op1 | op2;
42                 BITWISE_XOR:        aux1 = op1 ^ op2;
43                 LOGIC_EQUAL:        aux1 = {32{op1 == op2}};
```

```
44                  LOGIC_DIFFERENT:    aux1 = {32{op1 != op2}};
45                  LOGIC_LESS_THAN:    aux1 = {32{op1 < op2}};
46                  LOGIC_GREATER_THAN: aux1 = {32{op1 > op2}};
47          endcase
48  end
49
50  endmodule
```

4.2.7 NEXT JUMP CONTROLLER

A função deste bloco é definir o endereço da próxima instrução a ser executada; essa definição é feita com base no instructionPC, para interpretar o tipo de instrução, no PC atual, no caso de a próxima instrução ser relativa ao PC, regJmp, quando a próxima instrução é o valor absoluto contido em um registrador, e regCond, quando este desvio depende da condição contida em um registrador. O código NEXT JUMP CONTROLLER mostra como cada um desses tipos de instrução são interpretados e o que acontece em cada um deles.

Figura 37 – Bloco NEXT JUMP CONTROLLER



Fonte: O Autor

```
1   module nextJumpController
2   #(parameter TAM_PC = 16)
3   (
4           input wire  [31:0] instruction,
5           input wire  [(TAM_PC-1):0] PC,
6           input wire  [31:0] regJmp, regCond,
7           output wire [(TAM_PC-1):0] nextJump
8   );
9
10  reg [(TAM_PC-1):0] auxJump;
11  assign nextJump = auxJump;
12
```

```
13 reg [3:0] opcode;
14 reg [1:0] T;
15
16 always @ (regJmp or regCond or PC) begin
17
18         opcode = instruction[31:28];
19         T = instruction[1:0];
20
21         if (opcode == 4'b0110) begin
22                 case (T)
23                         2'b00: begin
24                                 auxJump = PC + instruction[(TAM_PC-1+2):2];
25                         end
26                         2'b01: begin
27                                 auxJump = instruction[(TAM_PC-1+2):2];
28                         end
29                         2'b10: begin // Jump para endereco contido no registrador
30                                 auxJump = regJmp[(TAM_PC-1):0];
31                         end
32                         2'b11: begin // Jump para endereco contido no registrador (
                               condicional)
33                                 auxJump = regCond == {32{1'b1}} ? regJmp[(TAM_PC-1):0] :
                                       PC + 1;
34                         end
35                         default: begin
36                                 auxJump = PC + 1;
37                         end
38                 endcase
39         end else begin
40                 auxJump = PC + 1;
41         end
42
43 end
44 endmodule
```

4.2.8 CONTROL UNIT

A função deste bloco é definir os valores de todos seletores a cada subida de *clock* de acordo com a Tabela 4. Os seletores definem coisas como as operações que serão realizadas e os tipos de dados que serão movimentados, por exemplo, o seletor selAlu define qual operação o bloco ALU deve realizar entre os operandos de entrada. O código CONTROL UNIT mostra como a definição dos seletores é feita.

Tabela 4 – Valores dos seletores por tipo de instrução.

Instrução	Subtipo	selTimer	selWrite EnableReg	selWrite EnableData	selOp2	selAlu	selWrite Type	selWrite	selSpecial	selSP
SUM	1	0	1	0	0	0000	00	10	00	0
	2	0	1	0	1	0000	00	10	00	0
SUBTRACT	1	0	1	0	0	0001	00	10	00	0
	2	0	1	0	1	0001	00	10	00	0
MULTIPLY	1	0	1	0	0	0010	01	10	00	0
	2	0	1	0	1	0010	01	10	00	0
DIVIDE	1	0	1	0	0	0011	01	10	00	0
	2	0	1	0	1	0011	10	10	00	0
SHIFT	1	0	1	0	1	0101	00	10	00	0
	2	0	1	0	1	0100	00	10	00	0
LOGIC	1	0	1	0	0	0110	00	10	00	0
	2	0	1	0	0	0111	00	10	00	0
	3	0	1	0	0	1000	00	10	00	0
	4	0	1	0	0	1001	00	10	00	0
	5	0	1	0	0	1010	00	10	00	0
	6	0	1	0	0	1011	00	10	00	0
	7	0	1	0	0	1100	00	10	00	0
	8	0	1	0	0	1101	00	10	00	0
JUMP	-	0	0	0	d	dddd	dd	dd	dd	0
STACK	1	0	0	1	d	dddd	00	00	00	1
	2	0	1	0	d	dddd	00	00	00	1
WRITE	-	0	1	0	d	dddd	dd	dd	dd	0
COPY	-	0	1	0	d	dddd	00	01	00	0
LOAD	-	0	1	0	d	dddd	00	00	00	0
STORE	-	0	0	1	d	dddd	dd	dd	dd	0
SLEEP	-	1	0	0	d	dddd	dd	dd	dd	0
INPUT	-	0	1	0	d	dddd	00	11	00	0

Fonte: O Autor

```
1  module controlUnit
2  (
3          input wire  [31:0] instruction,
4          output wire selTimer, selWriteEnableReg, selWriteEnableData, selOp2,
5          output wire [3:0] selALU,
6          output wire [1:0] selWriteType, selSpecial, selWrite,
7          output wire selSP,
8          input wire clk
9  );
10
11 parameter SUM                    = 4'b0000;
12 parameter SUBTRACT       = 4'b0001;
13 parameter MULTIPLY       = 4'b0010;
14 parameter DIVIDE                 = 4'b0011;
15 parameter SHIFT          = 4'b0100;
16 parameter LOGIC          = 4'b0101;
17 parameter JUMP                   = 4'b0110;
18 parameter STACK          = 4'b0111;
19 parameter WRITE          = 4'b1000;
20 parameter COPY                   = 4'b1001;
21 parameter LOAD                   = 4'b1010;
22 parameter STORE          = 4'b1011;
23 parameter SLEEP          = 4'b1100;
24 parameter INPUT          = 4'b1101;
25
26 reg regTimer, regWriteEnableReg, regWriteEnableData, regOp2, regSP;
27 reg [1:0] regWriteType, regWrite, regSpecial;
```

```
28 reg [3:0] regALU;
29
30 assign selTimer = regTimer;
31 assign selWriteEnableReg = regWriteEnableReg;
32 assign selWriteEnableData = regWriteEnableData;
33 assign selOp2 = regOp2;
34 assign selALU = regALU;
35 assign selWriteType = regWriteType;
36 assign selWrite = regWrite;
37 assign selSpecial = regSpecial;
38 assign selSP = regSP;
39
40 wire [3:0] opcode;
41
42 assign opcode = instruction[31:28];
43
44 always @ (posedge clk) begin
45
46         case (opcode)
47         SUM: begin
48             regTimer = 1'b0;
49             regWriteEnableReg = 1'b1;
50             regWriteEnableData = 1'b0;
51             regOp2 = instruction[0];
52             regALU = 4'b0000;
53             regWriteType = 2'b00;
54             regWrite = 2'b10;
55             regSpecial = 2'b00;
56                                 regSP = 1'b0;
57         end
58         SUBTRACT: begin
59             regTimer = 1'b0;
60             regWriteEnableReg = 1'b1;
61             regWriteEnableData = 1'b0;
62             regOp2 = instruction[0];
63             regALU = 4'b0001;
64             regWriteType = 2'b00;
65             regWrite = 2'b10;
66             regSpecial = 2'b00;
67                                 regSP = 1'b0;
68         end
69         MULTIPLY: begin
70             regTimer = 1'b0;
71             regWriteEnableReg = 1'b1;
72             regWriteEnableData = 1'b0;
73             regOp2 = instruction[0];
74             regALU = 4'b0010;
75             regWriteType = 2'b01;
76             regWrite = 2'b10;
77             regSpecial = 2'b00;
78                                 regSP = 1'b0;
79         end
80         DIVIDE: begin
81             regTimer = 1'b0;
82             regWriteEnableReg = 1'b1;
83             regWriteEnableData = 1'b0;
84             regOp2 = instruction[0];
85             regALU = 4'b0011;
86             regWriteType = 2'b10;
87             regWrite = 2'b10;
```

```
88              regSpecial = 2'b00;
89                                  regSP = 1'b0;
90          end
91          SHIFT: begin
92              regTimer = 1'b0;
93              regWriteEnableReg = 1'b1;
94              regWriteEnableData = 1'b0;
95              regOp2 = 1'b0;
96              regALU = instruction[0] == 1'b1 ? 4'b0100 : 4'b0101;
97              regWriteType = 2'b00;
98              regWrite = 2'b10;
99              regSpecial = 2'b00;
100                                 regSP = 1'b0;
101         end
102         LOGIC: begin
103             regTimer = 1'b0;
104             regWriteEnableReg = 1'b1;
105             regWriteEnableData = 1'b0;
106             regOp2 = 1'b0;
107             regALU = instruction[2:0] == 3'b000 ? 4'b0110 :
108                                                     instruction[2:0] == 3'b001 ? 4'
                                                        b0111 :
109                                                     instruction[2:0] == 3'b010 ? 4'
                                                        b1000 :
110                                                     instruction[2:0] == 3'b011 ? 4'
                                                        b1001 :
111                                                     instruction[2:0] == 3'b100 ? 4'
                                                        b1010 :
112                                                     instruction[2:0] == 3'b101 ? 4'
                                                        b1011 :
113                                                     instruction[2:0] == 3'b110 ? 4'
                                                        b1100 :
114                                                     4'b1101;
115             regWriteType = 2'b00;
116             regWrite = 2'b10;
117             regSpecial = 2'b00;
118                                 regSP = 1'b0;
119         end
120         JUMP: begin
121             regTimer = 1'b0;
122             regWriteEnableReg = 1'b0;
123             regWriteEnableData = 1'b0;
124             regOp2 = 1'b0;
125             regALU = 4'b0000;
126             regWriteType = 2'b00;
127             regWrite = 2'b10;
128             regSpecial = 2'b00;
129                                 regSP = 1'b0;
130         end
131         STACK: begin
132             regTimer = 1'b0;
133             regWriteEnableReg = 1'b0;
134             regWriteEnableData = instruction[0];
135             regOp2 = 1'b0;
136             regALU = 4'b0000;
137             regWriteType = 2'b00;
138             regWrite = 2'b10;
139             regSpecial = 2'b00;
140                                 regSP = 1'b1;
141         end
```

```
142         WRITE: begin
143             regTimer = 1'b0;
144             regWriteEnableReg = 1'b1;
145             regWriteEnableData = 1'b0;
146             regOp2 = 1'b0;
147             regALU = 4'b0000;
148             regWriteType = 2'b00;
149             regWrite = 2'b11;
150             regSpecial = 2'b00;
151                                 regSP = 1'b0;
152         end
153         COPY: begin
154             regTimer = 1'b0;
155             regWriteEnableReg = 1'b1;
156             regWriteEnableData = 1'b0;
157             regOp2 = 1'b0;
158             regALU = 4'b0000;
159             regWriteType = 2'b00;
160             regWrite = 2'b01;
161             regSpecial = instruction[1:0];
162                                 regSP = 1'b0;
163         end
164         LOAD: begin
165             regTimer = 1'b0;
166             regWriteEnableReg = 1'b1;
167             regWriteEnableData = 1'b0;
168             regOp2 = 1'b0;
169             regALU = 4'b0000;
170             regWriteType = 2'b00;
171             regWrite = 2'b00;
172             regSpecial = 2'b00;
173                                 regSP = 1'b0;
174         end
175         STORE: begin
176             regTimer = 1'b0;
177             regWriteEnableReg = 1'b0;
178             regWriteEnableData = 1'b1;
179             regOp2 = 1'b0;
180             regALU = 4'b0000;
181             regWriteType = 2'b00;
182             regWrite = 2'b10;
183             regSpecial = 2'b00;
184                                 regSP = 1'b0;
185         end
186         SLEEP: begin
187             regTimer = 1'b0;
188             regWriteEnableReg = 1'b0;
189             regWriteEnableData = 1'b0;
190             regOp2 = 1'b0;
191             regALU = 4'b0000;
192             regWriteType = 2'b00;
193             regWrite = 2'b10;
194             regSpecial = 2'b00;
195                                 regSP = 1'b0;
196         end
197                   INPUT: begin
198             regTimer = 1'b0;
199             regWriteEnableReg = 1'b1;
200             regWriteEnableData = 1'b0;
201             regOp2 = 1'b0;
```

```
            regALU = 4'b0000;
            regWriteType = 2'b00;
            regWrite = 2'b11;
            regSpecial = 2'b00;
                                regSP = 1'b0;
        end
        endcase

end

endmodule
```

4.2.9 Interligação entre os blocos

A implementação de todos os blocos foi interligada conforme o esquemático detalhado apresentado, formando o seguinte código que terá seu funcioamento demonstrado na próxima seção:

```
module project (
        input  clk,
        output wire [15:0] PC
);

parameter TAM_PC = 16;
parameter NOOP = 32'b0;
controlUnit(instruction, selTimer, selWriteEnableReg, selWriteEnableData, selOp2, selALU,
     selWriteType, selWrite, clk);

wire selTimer, selWriteEnableReg, selWriteEnableData, selOp2;
wire [2:0] selALU, selWriteType, selWrite;

wire [31:0] regJmp, regCond;
assign regJmp = regData1;
assign regCond = regData2;
nextJumpController NJC(instruction, PC, regJmp, regCond, nextJump);

wire [(TAM_PC-1):0] nextJump;
wire [(TAM_PC-1):0] nextPC;
assign nextPC = selTimer ? PC : nextJump;

regPC RPC(nextPC, PC, clk); //Atualizar PC

wire [(TAM_PC-1):0] PC;

programMemory PM(PC, instructionPC); // Mando PC, recebo instructionPC

wire [31:0] instructionPC;
wire [31:0] instruction;
wire [3:0]  opcode;
wire [4:0]  R1, R2, R3;
wire [31:0] writeData; // Informacao que sera guardada no endereco R3

assign instruction = selTimer ? NOOP : instructionPC;
assign opcode = instruction[31:28];
assign R3 = opcode == 4'b0010 || opcode == 4'b0011 || opcode == 4'b0110 || opcode == 4'
    b1011 ? instruction[17:13] : instruction[27:23];
```

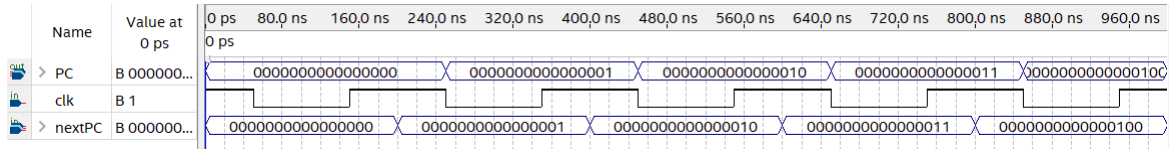
```
37 assign R1 = opcode == 4'b0010 || opcode == 4'b0011 || opcode == 4'b0110 || opcode == 4'
       b1011 ? instruction[27:23] : instruction[22:18];
38 assign R2 = opcode == 4'b0010 || opcode == 4'b0011 || opcode == 4'b0110 || opcode == 4'
       b1011 ? instruction[22:18] : instruction[17:13];
39
40 regFile(R1, R2, R3, regData1, regData2, writeData, selWriteType, selWriteEnableReg, clk);
41 immediateExtractor(instruction, immediate, clk);
42
43 wire [31:0] immediate;
44 wire [31:0] regData1, regData2;
45 wire [31:0] op1, op2;
46
47 assign op1 = regData1;
48 assign op2 = selOp2 ? immediate : regData2;
49
50 alu A(op1, op2, result, selALU);
51
52 // Ler memoria de dados
53 dataMemory DM(immediate, regData1, readData, selWriteEnableData, clk);
54
55 wire [63:0] result;
56 wire [31:0] readData, copyData;
57 wire [63:0] readDataExt, copyDataExt;
58
59 assign copyData = regData1;
60 assign readDataExt = {32'b0, readData};
61 assign copyDataExt = {32'b0, copyData};
62
63 assign writeData =
64         selWrite == 2'b00 ? readDataExt :
65         selWrite == 2'b01 ? copyDataExt :
66         selWrite == 2'b10 ? result : 64'b0;
67
68 endmodule
```

5 Resultados Obtidos e Discussões

A seguir serão apresentadas as simulações dos principais blocos pertencentes à unidade de processamento do sistema computacional que foram apresentados na seção 4.2, utilizando formas de onda geradas pelo *Quartus* que representam o funcionamento de cada bloco.

A simulação do bloco REG PC pode ser visualizada na Figura 38, onde é possúvel observar que é atribuido à PC o valor de nextPC sempre que ocorre uma descida de *clock*.

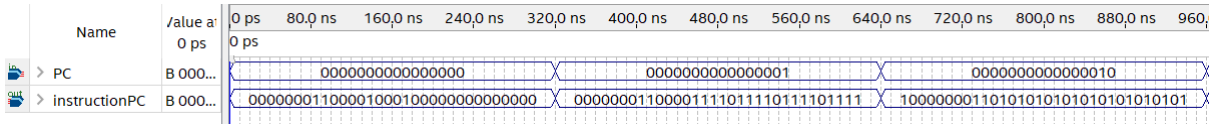
Figura 38 – Simulação do bloco REG PC



Fonte: O Autor

Na simulação realizada na Figura 39 sobre o bloco PROGRAM MEMORY, é possível observar que o valor de saída instructionPC é atualizado com base no valor de PC, que indica a posição da instrução que deve ser extraída da memória, ou seja, a respectiva instrução contida no arquivo "program.txt".

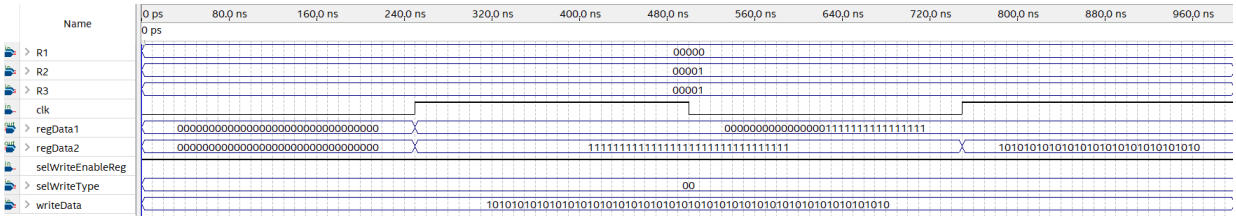
Figura 39 – Simulação do bloco PROGRAM MEMORY



Fonte: O Autor

O funcionamento do bloco REGISTER FILE foi simulado na Figura 40. Nessa figura podemos ver que os conteúdos armazenados nos registradores com índices R1 e R2 foram lidos e passados respectivamente para regData1 e regData2 na subida do *clock*. Já na descida do *clock*, como selWriteEnableReg estava habilitado, os 32 *bits* menos significativos de writeData foram armazenados no registrador de índice R3, que no exemplo é igual à R2 para que, na próxima subida de *clock*, possa ser visto que o valor do registrador foi realmente atualizado, como mostra a última etapa da figura.

Figura 40 – Simulação do bloco REGISTER FILE



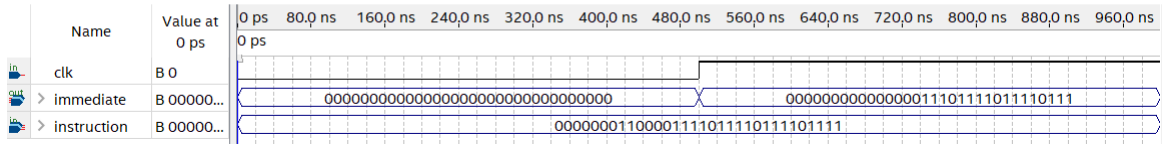
Fonte: O Autor

Para demonstrar o funcionamento do bloco IMMEDIATE EXTRACTOR, foram simulados dois exemplos de extração. O primeiro, ilustrado pela Figura 41(a), mostra a extração do imediato de uma instrução de soma entre registrador e imediato de 17 *bits*, apresentada na seção 4.1. O segundo, ilustrado pela Figura 41(b), mostra a extração do endereço de 23 *bits* da memória de dados de uma instrução do tipo write, também apresentada na seção 4.1.

Figura 41 – Simulação do bloco IMMEDIATE EXTRACTOR

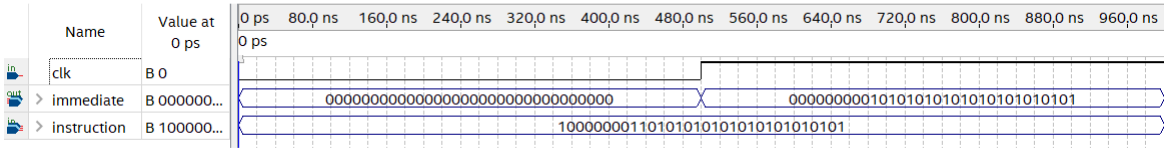
(a) Extração do imediato da instrução SOMA, subtipo 2

OPCODE	REG. OUT	REG. IN1	IMED. IN2	T
0000 (SOMA)	00011	00001	11101111011110111	1



(b) Extração do endereço da instrução WRITE

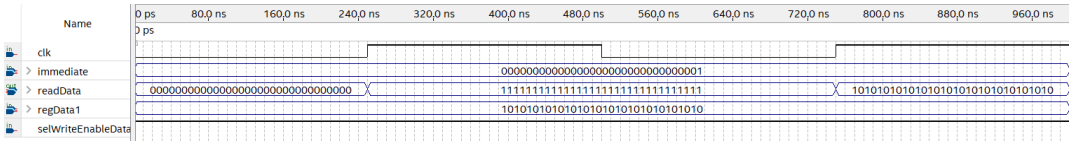
OPCODE	REG. OUT	IME. IN
1000 (WRITE)	00001	10101010101010101010101



Fonte: O Autor

O exemplo utilizado para demonstrar o funcionamento do bloco DATA MEMORY, ilustrado na Figura 42, é similar à simulação realizada para o bloco REGISTER FILE, onde dados são lidos na subida de *clock* e escritos na descida de *clock*, desde que o seletor selWriteEnableData esteja habilitado. No exemplo, 32 *bits* 1 começam armazenados no endereço especificado, depois de um ciclo completo de *clock*, descida para escrita do imediato e subida para leitura do novo valor, é possível observar que o valor do imediato foi armazenado na posição.

Figura 42 – Simulação do bloco DATA MEMORY

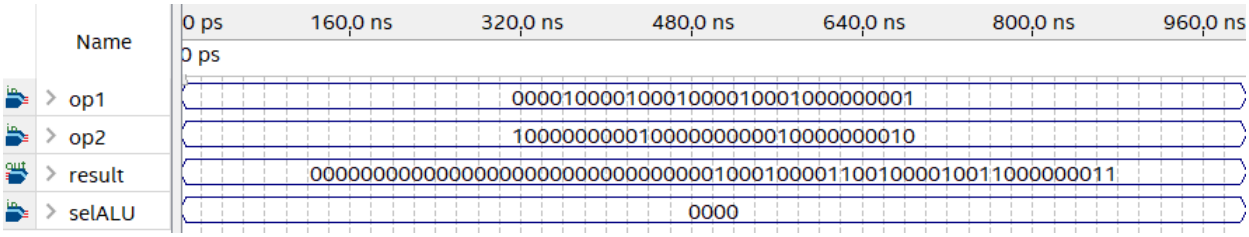


Fonte: O Autor

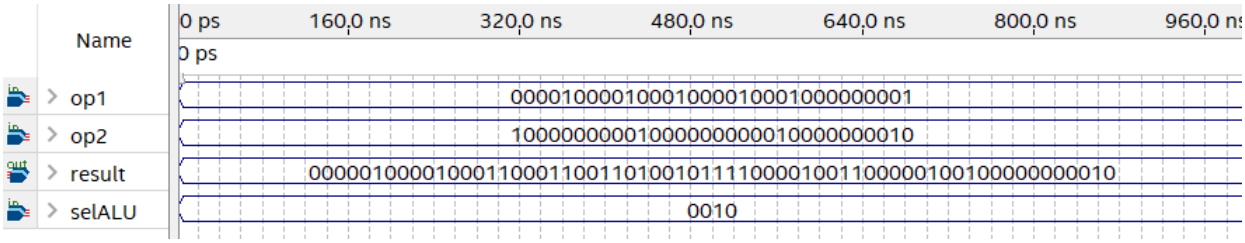
Para exemplificar o funcionamento do bloco ALU, foram realizadas duas operações aritméticas, primeiro, ilustrada na Figura 43(a), o resultado de 32 *bits* extendido da instrução soma, e segundo, Figura 43(b), o resultado de 64 *bits* da instrução multiplicação, ambas apresentadas na seção 4.1.

Figura 43 – Simulação do bloco ALU

(a) Soma



(b) Multiplicação



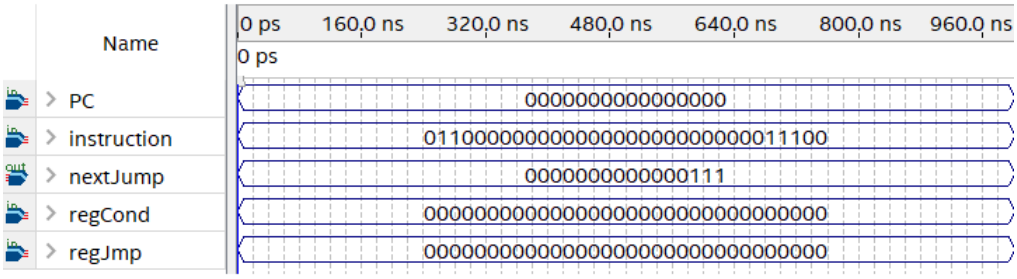
Fonte: O Autor

Foram realizadas duas simulações para mostrar o funcionamento do bloco NEXT JUMP CONTROLLER, uma para um desvio relativo ao PC usando a instrução JUMP subtipo 1, apresentada na seção 4.1 e referente à Figura 44(a), e outra para um desvio absoluto condicional usando a instrução JUMP subtipo 4, também apresentada na seção 4.1 e referente à **??**(b). No primeiro exemplo, é possível observar que o valor de PC é incrementado no valor presente no imediato da instrução. No segundo exemplo, regCond guarda o resultado que representa verdadeiro, então o valor de PC se torna o valor contido em regJmp, ou seja, 32 *bits* 1.

Figura 44 – Simulação do bloco NEXT JUMP CONTROLLER

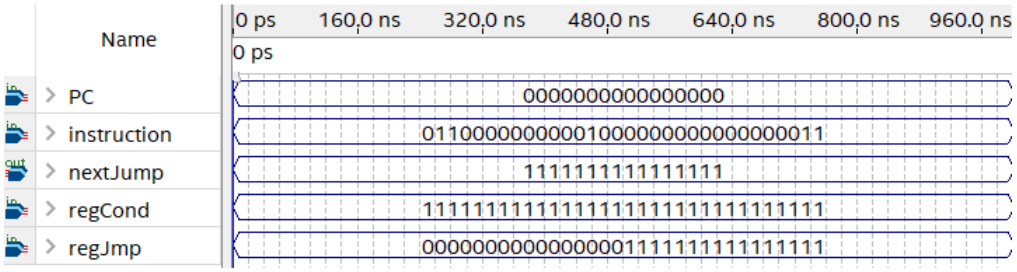
(a) Exemplo com JUMP subtipo 1

OPCODE	IME. IN	T
0110 (JUMP)	00000000000000000000000000000111	00



(b) Exemplo com JUMP subtipo 4

OPCODE	REG. JMP	REG. IN	NULL	T
0110 (JUMP)	00000	1	0000000000000000	11



Fonte: O Autor

Para demonstrar o funcionamento de todos os blocos interligados, a sequência de instruções apresentada na Figura 45 foi carregada no arquivo "program.txt", que é o arquivo inicializador do bloco PROGRAM MEMORY, como mostrado anteriormente. A sequência começa com a escrita dos valores em binário de 3 e 4 nos registradores de índice 0 e 1, respectivamente. Em seguida, esses dois valores são somados e o resultado 7 da soma é armazenado no registrador de índice 2. É realizado um JUMP de subtipo 3 tendo como destino a instrução com índice igual ao resultado da soma, ou seja, 7. A instrução 7 desvia o programa de volta para o fluxo anterior, partindo da instrução 4, que armazena no registrador 3 o resultaod da operação lógica "menor que"entre os registradores 0 e 1, ou seja, verdadeiro. Logo depois, o valor 8 em binário é armazenado no registrador 4 para que seja efetuado o JUMP condicional na instrução 6.

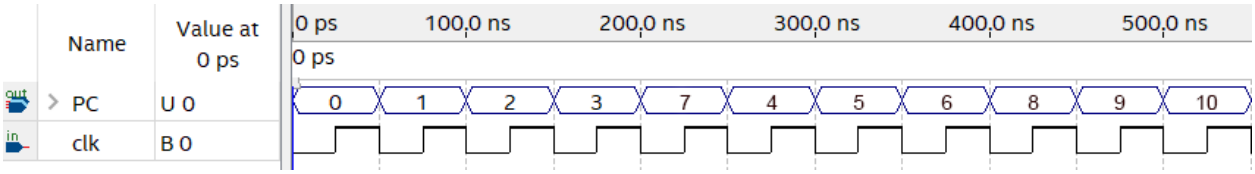
A simulação da Figura 46 mostra a sequência seguida por PC conforme as instruções são executadas, que corresponde exatamente à sequência esperada pela descrição acima, seguindo uma incrementação regular após o término das instruções (os valores de PC são exibidos em formato decimal para melhor visualização).

Figura 45 – Simulação dos blocos interligados - sequência de instruções

ÍNDICE	TIPO	SUBTIPO	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	WRITE	-	OPCODE				REG. OUT					IME. IN																							
			1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	WRITE	-	OPCODE				REG. OUT					IME. IN																							
			1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
2	SOMA	1	OPCODE				REG. OUT					REG. IN1					REG. IN2					NULL												T	
			0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	JUMP 7	3	OPCODE				REG. JMP					NULL																						T	
			0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
4	LÓGICA	8	OPCODE				REG. OUT					REG. IN1					REG. IN2					NULL												T	
			0	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1
5	WRITE	-	OPCODE				REG. OUT					IME. IN																							
			1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
6	JUMP 8	4	OPCODE				REG. JMP					REG. IN1					NULL																	T	
			0	1	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
7	JUMP 4	2	OPCODE				END. IN (PROGRAMA)																											T	
			0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

Fonte: O Autor

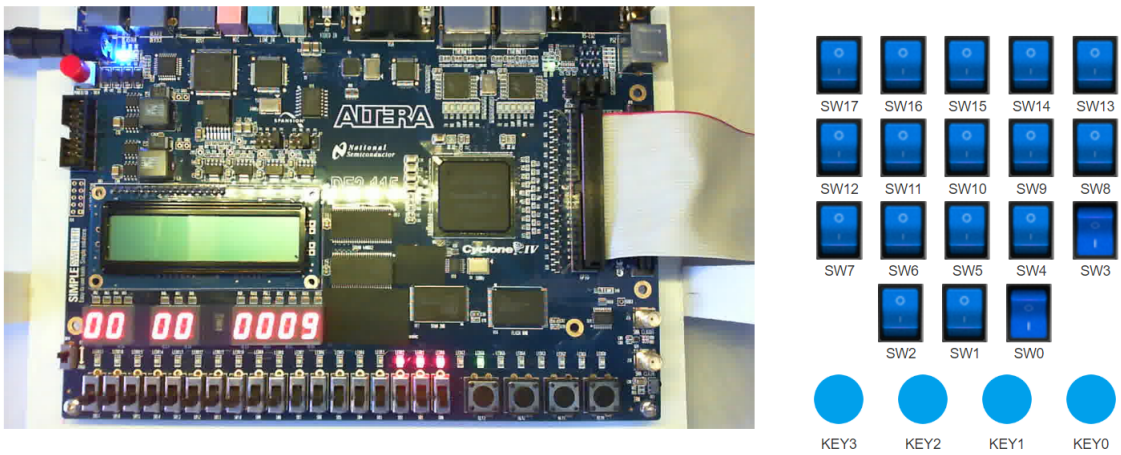
Figura 46 – Simulação dos blocos interligados - forma de onda



Fonte: O Autor

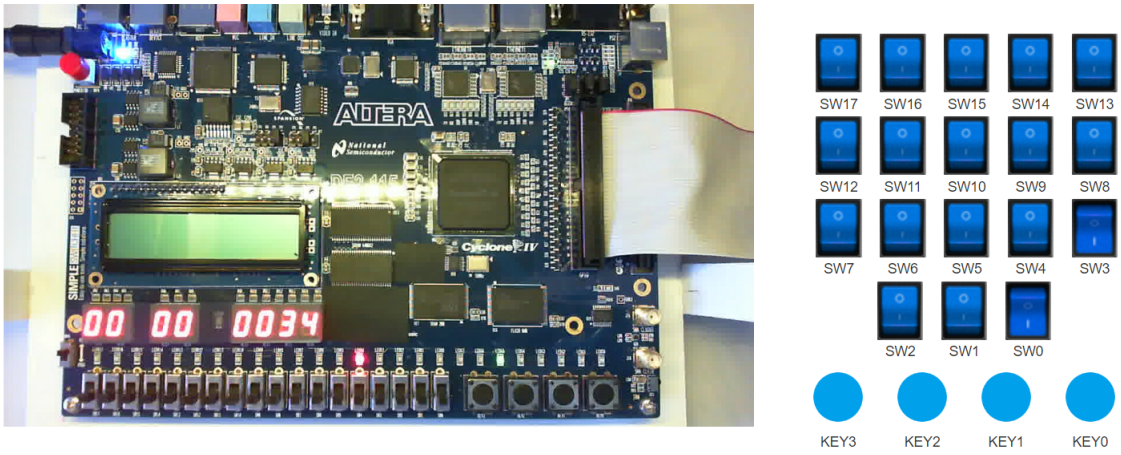
Com todos os blocos funcionando individualmente e interligados, foi realizada a integração com o dispositivo FPGA e em seguida realizados testes. Foi implementado um pseudocompilador para facilitar a elaboração dos algoritmos; os códigos serão apresentados em linguagem pré-compilação para facilitar o entendimento. O primeiro teste foi a elaboração e execução de um algoritmo que retorna o número da sequência de Fibonacci correspondente à posição solicitada. Como exemplo, foi fornecido como entrada o valor 9 (em binário), como mostra a Figura 47, resultando no valor 34, como mostra a Figura 48. O código fibonacci mostra o funcionamento do algoritmo.

Figura 47 – Simulação do algoritmo de Fibonacci - entrada



Fonte: O Autor

Figura 48 – Simulação do algoritmo de Fibonacci - saída



Fonte: O Autor

```
1  WRITE $0 @00000000000000000000000000
2  WRITE $1 @00000000000000000000000001
3  WRITE $2 @00000000000000000000000001
4  INPUT $10 NULL
5  COPY $31 $10 NULL @00
6  WRITE $30 @00000000000000000000001111
7  LOGIC $3 $2 $10 NULL @110
8  LOGIC $4 $2 $10 NULL @100
9  JUMP $30 $3 NULL @11
10 JUMP $30 $4 NULL @11
11 SUM $5 $0 $1 NULL @0
12 COPY $0 $1 NULL @00
13 COPY $1 $5 NULL @00
14 SUM $2 $2 @00000000000000001 @1
15 JUMP @00000000000000000000000110 @01
16 COPY $31 $1
17 INPUT $11 NULL
```

Como o algoritmo de Fibonacci não utiliza todos os tipos de instrução, foi executado um algoritmo sintético para testar o funcionamento do restante das instruções. O algoritmo sintético empilha e desempilha os resultados das operações SUBTRACT, SHIFT, MULTIPLY e DIVIDE, mostrando o funcionamento dessas instruções e das instruções do tipo STACK. No teste realizado no FPGA, foram fornecidos como entradas os valores 12 e 4, respectivamente, retornando os valores de saída 3 (divisão), 48 (multiplicação), 24 (um deslocamento à esquerda) e 8 (subtração), como mostra a Figura 49.

Figura 49 – Simulação do algoritmo sintético - saídas.

Fonte: O Autor

```
1  SUM $0 $0 NULL @1
2  INPUT $0 NULL
3  INPUT $1 NULL
4  SUBTRACT $2 $0 $1 NULL @0
5  STACK $2 NULL @0
6  SHIFT $3 $0 @00000000000000001 @0
7  STACK $3 NULL @0
8  MULTIPLY $0 $1 NULL @0
9  STACK $28 NULL @0
10 DIVIDE $0 $1 NULL @0
11 STACK $29 NULL @0
12 STACK $31 NULL @1
13 STACK $31 NULL @1
14 STACK $31 NULL @1
15 STACK $31 NULL @1
```

6 Considerações Finais

O objetivo do curso é desenvolver uma arquitetura computacional completa que seja capaz de resolver problemas computacionais básicos, começando pelo projeto e definição dos conceitos básicos de *design* e estrutura, como os componentes que serão usados e como os comandos serão executados. Nessa etapa está sendo contruído o alicerce de todo o sistema, onde serão desenvolvidas todas as partes principais do computador. A construção de uma base sólida e bem definida é essencial para o sucesso na evolução do sistema no futuro, e é isso que é mostrado nesse relatório: o projeto de uma estrutura que será capaz de sustentar um sistema computacional completo.

Referências

1 CLEMENTS, A. *The Principles of Computer Hardware*. 4th edition. ed. Oxford University Press, Inc., New York, NY: [s.n.], 2000. Citado na página 8.

2 CANALTI. *Arquitetura de Computadores (O que é, por que estudar)*. [S.l.], Acessado em 31/03/2019. Disponível em: <https://www.canalti.com.br/arquitetura-de-computadores/arquitetura-de-computadores-o-que-e-por-que-estudar/>. Citado na página 10.

3 GALDINO, D. J. *Aula 03 Organização de Computadores - Processadores - Introdução*. [S.l.], Acessado em 06/04/2019. Citado na página 10.

4 TOCCI, R. *Sistemas Digitais*. 10th edition. ed. [S.l.: s.n.], 2007. Citado na página 10.

5 FILHO, R. de G. N. *Fundamentos de Hardware*. [S.l.], Acessado em 06/04/2019. Disponível em: <http://www.di.ufpb.br/raimundo/ArqDI/Arq2.htm>. Citado 2 vezes nas páginas 10 e 11.

6 CANALTI. *Arquitetura e Organização de Computadores*. [S.l.], Acessado em 06/04/2019. Disponível em: <https://cadernogeek.wordpress.com/tag/conjunto-de-instrucoes/>. Citado na página 11.

7 STALLINGS, W. *Computer Organization and Architecture*. 8th edition. ed. Upper Saddle River, NJ 07458: [s.n.], 2009. Citado 2 vezes nas páginas 12 e 14.

8 AL, A. L. C. et. *Modos de endereçamento e conjunto de instruções*. [S.l.], Acessado em 06/04/2019. Citado 2 vezes nas páginas 12 e 13.

9 MONTEIRO, M. *Introdução à Organização de Computadores*. 5th edition. ed. Rio de Janeiro: LTC: [s.n.], 2012. Citado na página 12.

10 FLYNN, M. *Some computer organizations and their effectiveness*. IEEE Trans. Comput., vol. C-21, pp. 948-960: [s.n.], 1972. Citado na página 14.

11 INTEL QUARTUS PRIME DESIGN SOFTWARE. Breaking the Barriers of FPGA Design. Intel FPGA Development Tools. [S.l.], Acessado em 04/05/2019. Disponível em: <https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/overview.html?_ga=2.90972121.25176057.1544534037-954243754.1542224912&erpm_id=7000079>. Citado na página 14.

12 MIDORIKAWA, E. T. *Uma Introdução às Linguagens de Descrição de Hardware*. São Paulo, 2007. Citado na página 15.

13 FPGA. Altera DE2-115 Development and Education Board. 2010. [S.l.], Acessado em 04/05/2019. Disponível em: <https://www.ee.ryerson.ca/~courses/coe608/labs/DE2_115_User_Manual.pdf>. Citado na página 15.