Gabriel Kenji de Almeida

Implementação de um processador monociclo para o FPGA com o Quartus Prime: Implementação da unidade de processamento.

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

Abril de 2019

Resumo

O presente relatório tem por finalidade descrever como ocorreu o desenvolvimento e implementação da unidade de processamento na elaboração de um processador a ser simulado pelo FPGA (Field Programmable Gate Array), um chip reprogramável a nível de portas lógicas, por meio do uso do Verilog, uma linguagem de descrição de hardware. Todos os conceitos e métodos utilizados serão devidamente explicados posteriormente. A fim de complementar o primeiro relatório, este abordará algumas mudanças no caminho de dados, nos sinais de controle, e no banco de registradores, que ocorreram com a finalidade de fazer com que o processador possua a capacidade de executar todas as instruções propostas pelo conjunto de instruções escolhido. A implementação da unidade de processamento ocorreu de mondo gradativo, com a elaboração em Verilog de cada módulo do caminho de dados e, posteriormente, a inteligação deles. Todos os procedimentos e a funcionalidade de cada bloco serão apropriadamente esclarecidos.

Palavras-chaves: MIPS. Verilog. Processador. Quartus Prime.

Lista de ilustrações

Figura 1 – Ilustração do caminho de dados desenvolvido inicialmente, desenhado através do uso da ferramenta do site draw.io 20
Figura 2 – Última versão do caminho de dados modificado, desenhado através do uso da ferramenta do site draw.io 21
Figura 3 – Primeira simulação em forma de onda da Unidade de Lógica e Aritmética. 31
Figura 4 – Segunda simulação em forma de onda da Unidade de Lógica e Aritmética. 32
Figura 5 – Simulação em forma de onda realizada no banco de registradores. . . . 33
Figura 6 – Simulação em forma de onda da Memória de Dados. 33
Figura 7 – Simulação em forma de onda realizada na Memória de Instruções. . . . 34
Figura 8 – Simulação em forma de onda realizada no PC. 34
Figura 9 – Simulação em forma de onda realizada no PC. 34
Figura 10 – I 35

Lista de tabelas

Tabela 1 – Operações realizadas pela ULA 23

Sumário

1 INTRODUÇÃO 9

2 OBJETIVOS 11
2.1 Geral 11
2.2 Específico 11

3 FUNDAMENTAÇÃO TEÓRICA 13
3.1 A Arquitetura MIPS 13
3.2 Arquitetura base e conjunto de instruções. 13
3.3 Conjunto de Registradores 14
3.4 Sinais de Controle 14
3.5 Linguagem de descrição de hardware e o Verilog 14

4 DESENVOLVIMENTO 17
4.1 Conjunto de Registradores 17
4.2 Sinais de Controle 18
4.3 Mudanças no Caminho de Dados 20
4.4 Implementação 22

5 RESULTADOS OBTIDOS E DISCUSSÕES 31
5.1 Simulação da Unidade de Lógica e Aritmética 31
5.2 Simulação do Banco de Registradores 32
5.3 Simulação da Memória de Dados 33
5.4 Simulação da Memória de Instruções 33
5.5 Simulação do Contador de Programa 34
5.6 Simulação dos multiplexadores, do concatenador e do extensor . . . 34

6 CONSIDERAÇÕES FINAIS 37

REFERÊNCIAS 39

1 Introdução

O rápido avanço da tecnologia na última década certamente elevou a importância da eletrônica digital em um patamar nunca visto antes. É fato que a última geração da humanidade possui, em todo nível social e econômico, podendo até ser de modo indireto, uma certa dependência ou até comodismo proporcionados pela eletrônica digital.

Os sistemas digitais, no mundo atual, se tornaram onipresentes em todos os setores inerentes à sociedade. Como exemplo de alguns desses setores, tem-se a economia, a agricultura, a industria, o entretenimento, a cultura, entre outros.

Segundo Tocci, 2011(1), um sistema digital é uma combinação de dispositivos projetados para manipular informação lógica no formato digital, isto é, informações que podem assumir valores discretos. Esses dispositivos podem ser eletrônicos, mecânicos, magnéticos ou até pneumáticos. Entre os sistemas digitais mais conhecidos, temos os computadores, as calculadoras, os equipamentos de áudio e vídeo e o sistema de telecomunicações.

Um dos componentes presentes na maioria dos sistemas digitais mais complexos é o processador, que poderia ser entendido como sendo o "cérebro"do sistema, local onde todas as operações e tomadas de decisões são realizadas.

Sendo assim, o profissional atuante em áreas que se relacionam com a eletrônica digital deve dominar com maestria uma variedade diversificada de conceitos teóricos e práticos sobre sistemas digitais, incluindo conceitos relacionados a procesadores.

Um dos objetivos deste relatório, além do projeto de um processador funcional, foi expor esses conceitos de forma mais realista e prática, e sua realização tem um papel fundamental na formação profissional na área.

2 Objetivos

2.1 Geral

O projeto da disciplina Laboratório de Arquitetura e Organização de Computadores tem como objetivo principal a elaboração e o desenvolvimento de um processador funcional a partir de uma arquitetura customizada baseada no *MIPS*, que possua a capacidade de executar os algoritmos mais básicos da literatura de lógica de programação. A implementação ocorrerá por meio do uso da linguagem de descrição de hardware Verilog, no ambiente do software Quartus II. O processador será simulado no dispositivo FPGA (2) DE2-115 Cyclone IV.

2.2 Específico

A segunda etapa do projeto, como foi definido pelo Ponto de Checagem 2, tem como finalidade a implementação de toda a unidade de processamento do sistema. Sendo assim, pontuando cada aspecto trabalhado, tem-se:

- Adição de alguns registradores específicos no banco de registradores;
- Adição de outros sinais de controle;
- Remoção e criação de novos módulos, elaborando um novo caminho de dados;
- Implementação individual de cada módulo;
- Simulação dos módulos implementados, para verificar se funcionam de modo correto.

Vale ressaltar que essas mudanças foram realizadas para que o processador venha a realizar adequadamente todas as instruções estipuladas no Primeiro ponto de Checagem.

3 Fundamentação Teórica

Nesta seção, buscou-se discorrer sobre os elementos teóricos e conceituais necessários no desenvolvimento dos objetivos específicos, discutidos no capítulo anterior.

3.1 A Arquitetura MIPS

A principal arquitetura utilizada como modelo ou referência no desenvolvimento do projeto foi a arquitetura MIPS que significa Microprocessor without interlocked pipeline stages, ou Microprocessador sem estágios intertravados de pipeline. Se trata de um microprocessador RISC, cuja CPU utiliza apenas registradores para realizar operações lógicas e aritméticas.

3.2 Arquitetura base e conjunto de instruções.

Antes de definir o conjunto de instruções, a decisão de projeto inicial nessa implementação deve ser o tipo de armazenamento interno do processador.Segundo Patterson, 2013(3), as principais opções dessa parte da arquitetura seriam o armazenamento por pilha, por acumulador, ou por um conjunto de registradores. Vale ressaltar que nas duas primeiras opções, os operandos são nomeados de forma implicíta. Na primeira, o operando está implicitamente no topo da pilha, e na segunda, o operando é o acumulador. Como a escolha de um conjunto de registradores faz parte dos objetivos específicos, nota-se que a arquitetura deste projeto será do tipo registrador-registrador, correspondente a terceira opção. Nesta arquitetura, todos os operandos são explícitos, sendo registradores ou posições na memória.

Também conhecido como código de máquina(4), o conjunto de instruções comporta todas as operações, sejam elas complexas ou simples, que o processador pode vir a executar. Os conjuntos de instruções são comumente classificados como RISC (Reduced Instruction Set Computer) ou CISC (Complex Instruction Set Computer). As instruções escolhidas para compor o conjunto de instruções utilizado e suas particularidades serão esclarecidas no próximo capítulo.

Segundo Patterson(3), um conjunto de instruções do tipo CISC geralmente utiliza uma arquitetura do tipo Registrador-Memória, possui um número elevado de instruções distintas, complexas e mais específicas, e acessa os dados via memória.

Um conjunto de instruções do tipo RISC, que foi o utilizado nesse projeto, apresenta uma arquitetura do tipo registrador-registrador, com um número baixo de instruções

simples. A proposta desse tipo de conjunto é realizar tarefas mais complexas utilizando operações simples. O acesso aos dados ocorre por meio de registradores.

Vale ressaltar que uma das decisões de projeto sobre a arquitetura base tomadas nas primeiras etapas foi seguir o modelo de Arquitetura Havard, que consiste basicamente no uso de duas memórias, uma contém as instruções a serem executadas e outra contém os dados.

3.3 Conjunto de Registradores

Um dos módulos fundamentais do processador desenvolvido é o banco de registradores, que consiste em um conjunto de registradores responsável por armazenar os dados com os quais o processador trabalha. Durante o funcionamento do processador, esse módulo conversa diretamente com a memória de dados e de instruções, para atualizar o conteúdo dos registradores afim de satisfazer as necessidades da instrução executada. Nesse projeto, os registradores utilizado neste módulo apresentam funções e utilidades particulares, que serão devidamente esclarecidas no próximo capítulo.

3.4 Sinais de Controle

O módulo mais fundamental no projeto do processador é a unidade de controle, responsável por, a partir da instrução executada no momento, controlar o acionamento e o modo de operação dos demais módulos. Para que isso ocorra, cada módulo é ligado a unidade de controle por um barramento, que carregam os sinais de controle. Sendo assim, um sinal de controle é o modo com o qual a unidade de controle manuseia os outros módulos. Os sinais de controle escolhidos nesse projeto apresentam funcionalidade semelhante aos utilizados pelo modelo MIPS, com algumas ressalvas e adições, que serão devidamente esclarecidas no capítulo de desenvolvimento.

3.5 Linguagem de descrição de hardware e o Verilog

Uma HDL (Hardware Descriptive Language), ou, em português, linguagem de descrição de hardware, é usada exclusivamente para projetos de sistemas digitais. Sua principal particularidade em relação com linguagens de programação tradicionais é o paralelismo. Em geral, uma HDL possui diversas semelhanças com linguagens de programação, mas, como os circuitos digitais operam de forma paralela, eles não podem ser descritos por uma linguagem de programação tradicional, que opera de modo sequencial.

Entre as características mais relevantes da HDL, pode-se citar:

- A possibilidade de simplificação de circuitos digitais complexos através de poucas linhas de código;
- Reutilização de bibliotecas e projetos já implementados
- O código é independente do hardware. Sendo assim, existe a possibilidade de um mesmo código ser compatível com diferentes tipos de dispositivos;
- Possue um conjunto de regras de sintaxe bem semelhante ao utilizado por linguagens de programação tradicionais.

O Verilog é uma HDL usada para modelar sistemas eletrônicos a nível de circuito, semelhante com a proposta de toda HDL. Essa ferramenta, por sua vez, suporta projeção, verificação e implementação de projetos analógicos, digitais e híbridos em vários níveis de abstração. Com placas de desenvolvimento baseadas nos Circuitos Integrados específicos como, no caso deste trabalho, o FPGA, é possível descarregar o código gerado nessa linguagem para matrizes de portas lógicas combinacionais e sequenciais.

Esta linguagem difere das outras, em especial, pela maneira como é executada. Ela segue padrões diferentes das demais linguagens. Uma implementação em Verilog separa hierarquicamente os módulos de conexões e registradores. Sendo assim, é possível dividir o programa em processos sequenciais e paralelos, com circuitos combinacionais ou sequenciais. Processos sequenciais são executados dentro de blocos "begin/end". Os demais processos são executados de forma paralela. Os circuitos combinacionais são implementáveis através de atribuições do tipo bloqueante, e os circuitos sequenciais através de atribuições do tipo não-bloqueante. Os aspectos de sintaxe dessa linguagem são irrelevantes no entendimento do processo adotado nesse projeto. O único ponto que vale ressaltar é a possilidade de uso de todas as operações lógicas e aritméticas.(5)

4 Desenvolvimento

As informações expostas neste capítulo possuem a finalidade de apresentar e esclarecer todas as mudanças conceituais realizadas no projeto, e evidenciar suas necessidades e consequências, assim como apresentar a implementação realizada de todos os módulos da unidade de processamento. Antes de tudo, a primeira mudança realizada foi a inclusão de duas novas instruções no conjunto de instruções, para haver possibilidade do processador trabalhar com entrada e saída de dados, e uma nova instrução utilizada para deixar o processador parado:

- **in**: Faz com que um registrador específico, do banco de registradores, armazene um valor definido pelas chaves do dispositivo FPGA.
- **out**: Faz com que o conjunto de *displays* de 7 segmentos do dispositivo FPGA exiba um dado armazenado em um registrador específico do banco de registradores.
- **break**: Faz com que o processador fique em estado de *stand-by*, sem executar nenhuma instrução.

É importante ressaltar que não há necessidade de atribuir um formato a essas duas novas instruções, uma vez que o único campo relevante destas é o **Opcode**.

4.1 Conjunto de Registradores

Em relação ao conjunto de registradores adotado anteriormente, este novo possui dois novos registradores, relacionados ao procedimento dos dispositivos de entrada e saída, que será adequadamente esclarecido nas próximas seções. Ao todo, serão 34 registradores, sendo que 33 deles estarão presentes no módulo denotado por Banco de Registradores. Entre os registradores utilizados, tem-se:

- Os registradores High(HI) e Low(LO), utilizados para armazenar a parte mais significativa e a menos significativa de uma multiplicação, ou o resto e o quociente de uma divisão, respectivamente.
- 10 registradores temporários.
- 10 registradores de uso geral.
- 5 registradores que armazenam argumentos para funções.
- 2 registradores que armazenam resultados das funções

- Um registrador de endereço ($ra), utilizado para armazenar o endereço de retorno da instrução **jal**.
- Um registrador de pilha, que armazena o endereço do topo de uma pilha na memória.
- Um registrador para armazenm ar o endereço da próxima instrução a ser executada. Este registrador não está no banco de registradores. No diagrama em bloco, ele é representado pelo módulo com o rótulo "PC"(Program Counter).
- Um registrador de saída, que, com o módulo de saída, que será apresentado posteriormente, envia um dado para ser exibido nos *displays* de 7 segmentos do dispositivo FPGA utilizado.
- Um registrador de entrada, que, em conjunto com o módulo de entrada, apresentado posteriormente, armazena um dado gerado a partir das posições das chaves do dispositivo FPGA.

4.2 Sinais de Controle

Os novos sinais de controle foram incluídos para tratar as duas novas instruções, que, como ja foi dito, trabalham com os procedimentos de entrada e saída do processador. Todos os sinais de controle utilizados estão listados a seguir:

- **RegDst**: Se trata de um sinal de 2 bits que opera de 3 modos diferentes: se os 2 bits estão em 0, a entrada "Registrador para Escrita", do banco de registradores, que poderá ser visualizada no diagrama em bloco da próxima seção, recebe o campo R2 do código de instrução. Se o primeiro bit está em 0 e o segundo em 1, A mesma entrada recebe o campo R1 do código de instrução. Se a primeira entrada está em 1 e a segunda em 0, a entrada citada anteriormente recebe o endereço do registrador $ra, exclusivo para uso da instrução **jal**.
- **EscreveReg**: Quando inativo, não faz nada. Quando ativo, o registrador apontado pelo endereço contido no registrador "Registrador para escrita"é escrito com o valor de "Dados para escrita".
- **OrigALU**: Quanto inativo, o segundo operando da ALU vem da segunda saída do banco de registradores. Quando ativo, o segundo operando da ALU consiste nos 16 bits menos significativos da instrução, que correspondem ao campo "Imediato", com sinal extendido.
- **OrigPC**: Quando inativo, o PC é substituído pela saída do somador, que calcula o valor da próxima instrução. Quando ativo, o PC é substituído pela saída do somador que calcula o desvio.

- **LeMem**: Quanto inativo, não faz nada. Quando ativo, o conteúdo da memório de dados designado pela entrada "Endereço"é colocado na saída de "Dados da leitura".

- **EscreveMem**: Quando inativo, não faz nada. Quando ativo, o conteúdo da memória de dados designado pela entrada "Endereço"é substituído pelo valor na entrada "Dados para escrita".

- **MemparaReg**: Quando inativo, o valor enviado para a entrada "Dados para a escrita"do banco de registradores vem da ALU. Quando ativo, o valor enviado para a entrada de memória vem da memória de dados.

- **OpALU**: Assim como o primeiro sinal apresentado, também possui 2 bits e 3 modos de operação. Com os dois sinais em 0, para instruções de transferência de informação, a ALU realiza uma adição. Se o primeiro bit assumir o valor 0 e o segundo assumir 1, a ALU realiza uma subtração, para instruções de comparação. Se o primeiro bit assumir o valor 1 e o segundo assumir 0, a operação da ALU será determinada pela Unidade de controle da ALU, outro módulo presente no processador que possui a finalidade de determinar a operação a ser realizada pela ALU a partir do campo funct do código de instrução.

- **WriteRa**:Quando inativo, o campo "Dados para escrita"do banco de registradores recebe o resultado da ALU. Quando ativo, esse mesmo campo recebe o endereço atual do PC. Esse sinal é especifico para instruções **jal**.

- **BifNot**: Ativo quando uma instrução **bne** é executada, sinalizando que, caso a condição do desvio seja satisfeita, o PC irá receber o endereço calculado.

- **Branch**: Análogo ao sinal anterior, porém para a instrução **beq**.

- **Jump**: Quando inativo, o registrador PC recebe o endereço da próxima instrução ou o endereço do desvio condicional. Quando ativo, o registrados PC recebe o endereço de desvio incondicional de uma instrução do tipo. Esse sinal sempre é ativo quando uma instrução de salto incondicional é realizada **J**.

- **JumpR**: Esse sinal é sempre ativo quando uma instrução **jr** é executada. Quando inativo, caso uma instrução de salto incondicional seja realizada, ela acontece utilizando os 26 bits menos significativos da instrução. Quando ativo, caso esse tipo de instrução seja realizado, utiliza-se os 26 bits menos significativos do registrador **$ra**;

- **In**: Esse sinal é sempre ativo quando uma instrução **in** é executada. Quando ativo, este sinal faz com que um registrador específico do banco de registradores receba os dados gerados pelo módulo de entrada, que será apresentado na próxima seção;

- **Out**: Esse sinal é sempre ativo quando uma instrução **out** é executada. Quando ativo, faz com que o valor de um registrador específico do banco de registradores seja enviado para o módulo de saída, que também sera apresentado na próxima seção.

- **break**: Este sinal, que recebeu o mesmo nome da instrução, quando ativo, faz com que o processador não realize mais nenhuma instrução.

4.3 Mudanças no Caminho de Dados

As figuras 1 e 2 ilustram, respectivamente, o caminho de dados inicial e o modificado. Note que o segundo é mais denso e robusto. É fato atende todas as particularidades do conjunto de instruções, mas com o preço do aumento de complexidade, dificultando a compreensão.

Figura 1 – Ilustração do caminho de dados desenvolvido inicialmente, desenhado através do uso da ferramenta do site draw.io

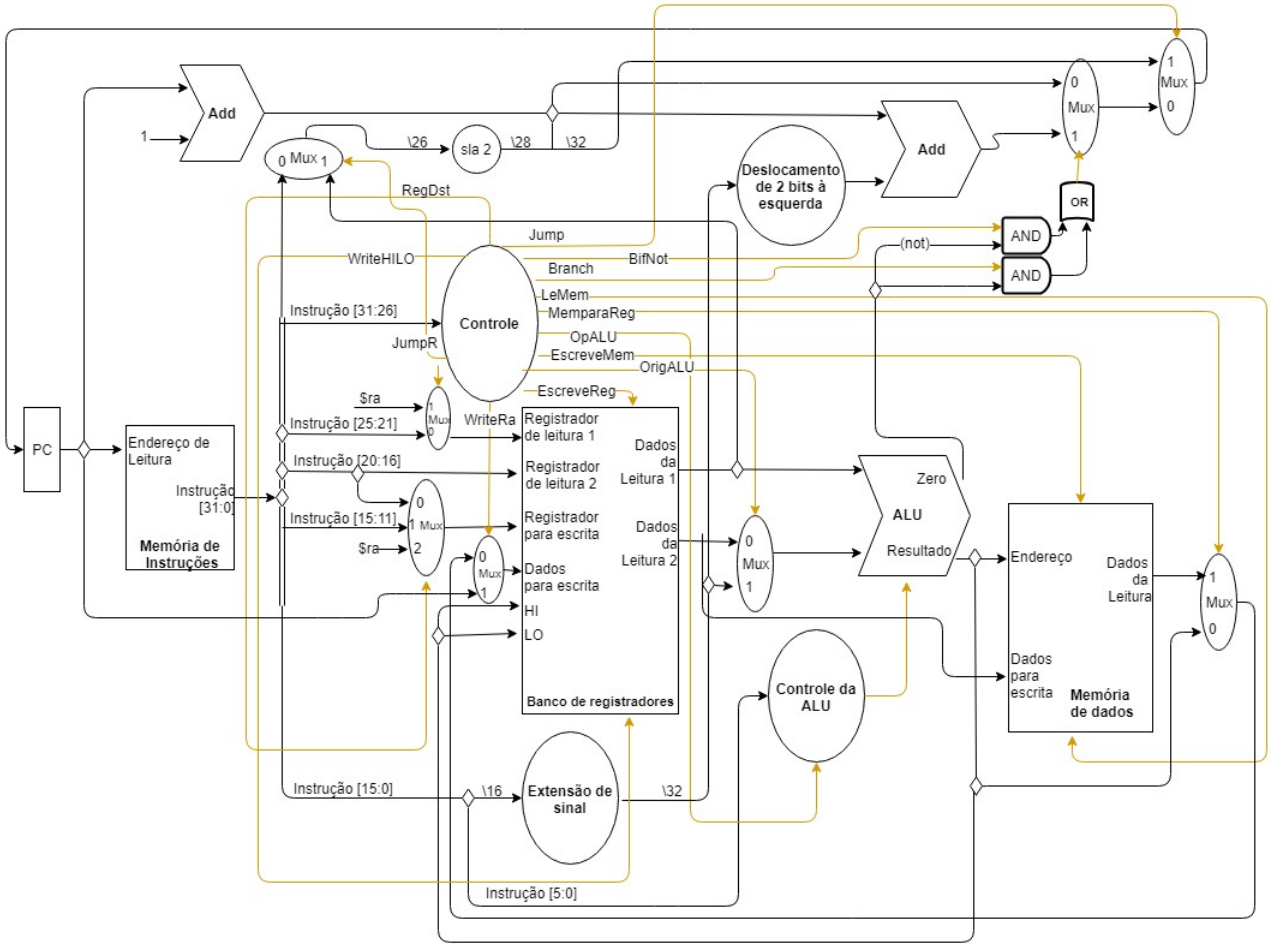
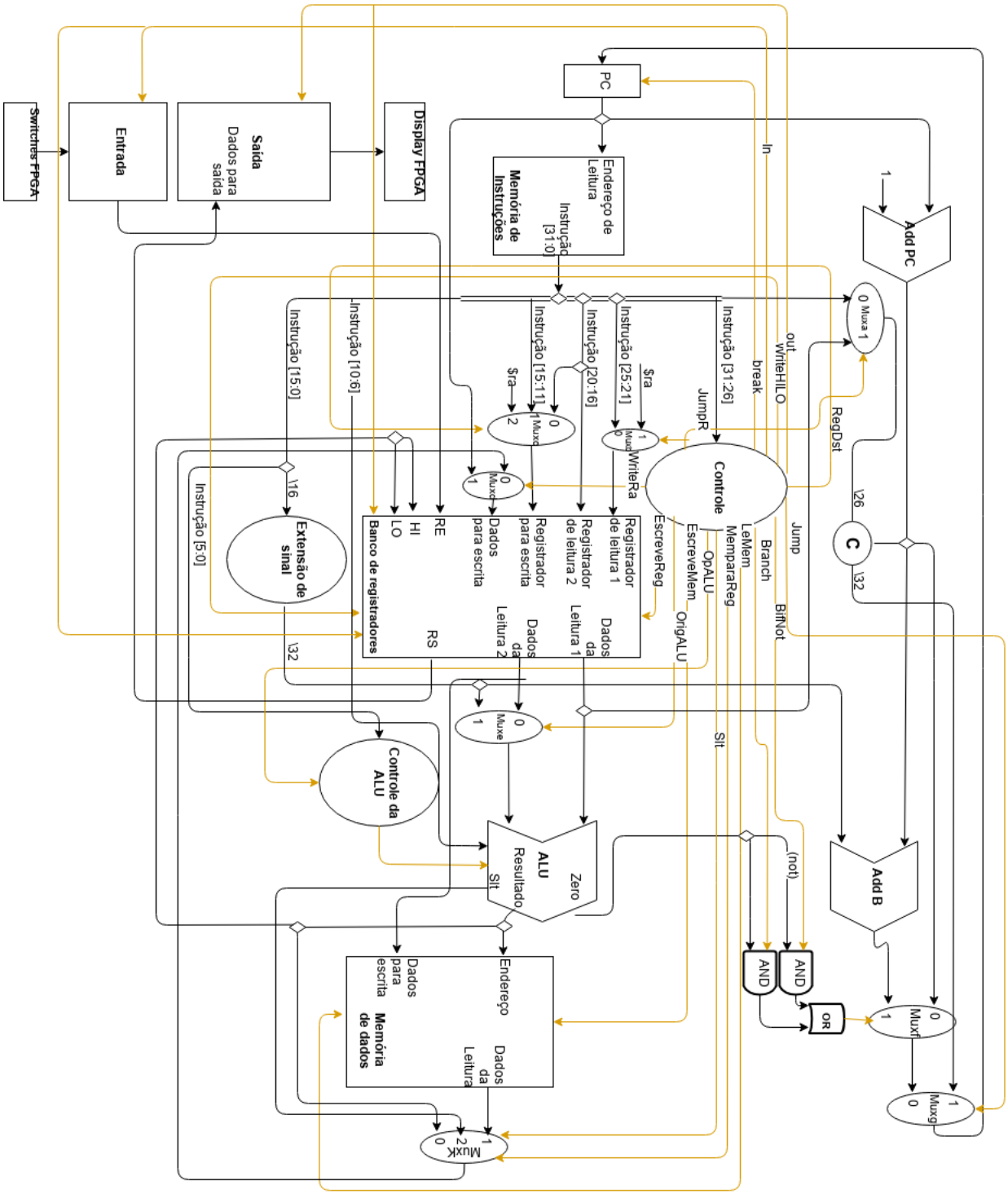


Figura 2 – Última versão do caminho de dados modificado, desenhado através do uso da ferramenta do site draw.io



As principais mudanças realizadas, como se pode observar, foram a inclusão dos módulos de entrada e de saída, a ligação de um novo barramento na Unidade de Lógica e Aritmética, a remoção dos dois módulos de deslocamento de 2 bits, utilizados em saltos, e a inclusão de um módulo denotado por "C".

O novo barramento utilizado na ULA é responsável por passar como parâmetro a parcela da instrução executada que corresponde ao deslocamento, para que a ULA faça as operações adequadas, com os operandos corretos, no caso de instruções de deslocamento.

Os procedimentos de entrada e saída, que são trabalhados de maneiras distintas, utilizam os módulos de entrada e de saída em conjunto com o banco de registradores e com os sinais de controle específicos, para operar a entrada e saída externa de dados, através dos *displays* e dos *switches*, nativos da placa FPGA, que são denotados no caminho de dados por dois outros blocos.

Os módulos de deslocamento, presentes na figura 1, um deles denotado por "sla"*(Shift Left Arithmetical 2)*, em um processador MIPS convencional, são utilizados em razão do fato da memória ser mapeada por palavras de 8 bits. Como o processador trabalha com 32 bits, um salto de, por exemplo, 5 posições deveria "pular"20 posições na memória de de 8 bits. Por isso, é necessário multiplicar o valor do salto por 4, ou seja, deslocar o número binário duas vezes para a esquerda. Como, neste projeto, a memória é mapeada com palavras de 32 bits, esse procedimento não é necessário.

O módulo **C** possui a função de concatenar os 26 bits de endereço do salto, no caso de uma instrução do tipo J, com os 6 bits mais significativos do **PC**. Esse procedimento não é necessariamente importante no caso do processador desenvolvido, pelo fato de ser mais simples, mas, em um processador funcional e robusto, esse procedimento faz com que os saltos possam atingir endereços maiores que 26 bits e que sejam realizados em endereços "próximos"ao valor do **PC**.

4.4 Implementação

Nesta seção haverá a descrição da implementação de cada módulo da unidade de processamento, e como ocorreu seu desenvolvimento. Primeiramente, será apresentado o código desenvolvido para o módulo **PC**. Como dito anteriormente, o **PC** é um registrador, porém, com algumas particularidades: Possui um valor inicial, que contém o endereço para a primeira posição da memória de instruções, botão *reset* e a entrada para o sinal de controle **break**, que possui a capacidade de travar o valor do **PC**.

```
1  module Program_counter (breakk, entrada_PC, saida_PC, clk, reset);
2
3          input breakk, clk, reset;
4          input [31:0] entrada_PC;
5
6          output reg [31:0] saida_PC;
7
8          initial begin
9          saida_PC = 1;
10         end
11
12         always @(posedge clk)
```

```
13          begin
14
15                  if(breakk == 0) saida_PC = entrada_PC;
16                  if(reset) saida_PC = 1;
17
18          end
19
20  endmodule
```

Observe que as funcionalidades descritas são implementadas apropriadamente pelo código apresentado.

A ULA *(Unidade de Lógica e Aritmética)*, como será possível de se visualizar a seguir, possui entradas que denotam os operandos, o deslocamento, em caso de operações de deslocamento, e o sinal de controle, que consiste em um barramento de 4 entradas. Como saídas, tem-se o resultado da operação e dois sinais de um bit, denotados por "slt"e "zero"Como o próprio nome denota, essa unidade pode ser entendida como a "calculadora"do processador, uma vez que é neste módulo em que todas as operações lógicas e aritméticas são realizadas. O barramento citado anteriormente possui por finalidade indicar a operação a ser realizada, de acordo com a seguinte tabela:

Tabela 1 – Operações realizadas pela ULA

Código	Operação
0000	—
0001	Adição
0010	Subtração
0011	Multiplicação
0100	Divisão
0101	Deslocamento a esquerda
0110	Deslocamento a direita
0111	And
1000	Or
1001	Xor
1010	Nor
1011	Not

A saída denotada no código por "shift"corresponde ao deslocamento a ser realizado na ULA caso a instrução seja o calculo de um deslocamento. As saídas "zero"e "slt"são ativadas quando a operação realizada é de subtração, e o resultado é 0 e negativo, respectivamente. Essas saídas são utilizadas para verificar a condição das instruções de salto condicional e do tipo *"Set Less Than"*. A saída denotada por "Resultado"possui 64 bits, com a finalidade de tratar os casos em que uma escrita nos registradores "HI"e "LO"é necessária. Nesses casos, os 32 bits mais significativos representarão os dados a serem escritos no registrador "HI"e os 32 menos significativos representarão os que serão escritos no registrador "LO".

```
1  module processador (Resultado, slt, zero, codigo_entrada, operador1, operador2, shift);
2
```

```
3         output reg [63:0] Resultado;
4         output reg slt, zero;
5
6         //reg[63:0] Resultado_reg;
7         //reg slt_reg, zero_reg;
8         reg[31:0] HIdiv;
9         reg[31:0] LOdiv;
10
11        input [3:0] codigo_entrada;
12        input [31:0] operador1;
13        input [31:0] operador2;
14        input [4:0] shift;
15
16        always @(*)
17        begin
18
19                case(codigo_entrada)
20
21                        4'b0001: Resultado = operador1 + operador2;
22
23                        4'b0010:
24                                begin
25                                Resultado= operador1 - operador2;
26                                slt = Resultado < 0 ? 1:0 ;
27                                zero = Resultado == 0 ? 1:0 ;
28                                end
29
30                        4'b0011: Resultado = operador1 * operador2;
31
32                        4'b0100:
33                                begin
34                                HIdiv = operador1 % operador2;
35                                LOdiv = operador1 / operador2;
36                                Resultado = {HIdiv, LOdiv};
37                                end
38
39                        4'b0101: Resultado = operador1 << shift;
40
41                        4'b0110: Resultado = operador1 >> shift;
42
43                        4'b0111: Resultado = operador1 & operador2;
44
45                        4'b1000: Resultado = operador1 | operador2;
46
47                        4'b1001: Resultado = operador1 ^~ operador2;
48
49                        4'b1010: Resultado = ~(operador1 | operador2);
50
51                        4'b1011: Resultado = ~operador1;
52
53                endcase
54        end
55
56
57
58  endmodule
```

O banco de registradores apresenta uma complexidade relativamente maior em relação aos outros módulos, que ocorre devido ao alto número de entradas e de procedi-

mentos. Para atender as necessidade do processador, esse módulo é capaz de realizar duas leituras simultaneamente, enquanto que realiza apenas uma escrita por ciclo de *clock*.

Como saída deste módulo, tem-se os dados da primeira e da segunda leitura, e os dados de saída, denotados por RS, utilizados para exibir informações nos *displays*, como já foi explicado anteriormente. Como entradas, tem-se:

- Dois endereços de leitura, para a função de leitura do banco de registradores;
- Um endereço de escrita, um barramento com os dados a serem escritos e o sinal de controle "EscreveReg". Quando o sinal é ativo, uma escrita é realiazada.
- Para a instrução de entrada de dados, tem-se os dados a serem escritos, denotados por "Dados_entrada"e o sinal de controle "In", que, quando ativo, faz com que o registrador de entrada seja escrito com os dados contidos em "Dados_entrada".
- Para a instrução de saída de dados, tem-se o sinal de controle "In", que faz com que o registrador "RS"seja gravado com o valor do barramento com os dados a serem escritos.
- Para os casos em que os registradores "HI"e "LO"precisam ser escritos simultaneamente, tem-se a entrada denotada por "Resultado", que é o resultado da operação calculada pela ULA, e o sinal de controle "WriteHILO"que, quando ativo, faz com que os registradores sejam escritos com o valor de "Resultado". Os 32 bits mais significativos são escritos em "HI"e os 32 menos significativos em "LO".

```
1  module banco_Registradores (Reg_leitura1, Reg_leitura2, Reg_escrita, Dados_entrada,
       Dados_escrita, Dados_leitura1,
2  Dados_leitura2, RS, EscreveReg, WriteHILO, In, ResultadoHILO, clk, Out);
3
4  parameter RA = 5'b11111;
5  parameter HI = 5'b11110;
6  parameter LO = 5'b11101;
7  parameter RE = 5'b11100;
8
9  input [4:0] Reg_leitura1;
10 input [4:0] Reg_leitura2;
11 input [31:0] Dados_escrita;
12 input [31:0] Dados_entrada;
13 input clk;
14 input EscreveReg, WriteHILO, Out, In;
15 input [63:0] ResultadoHILO;
16 input [4:0] Reg_escrita;
17
18 output wire [31:0] Dados_leitura1;
19 output wire [31:0] Dados_leitura2;
20 output reg [31:0] RS;
21
22 reg [31:0] registradores_banco [31:0];
23
24 assign Dados_leitura1 = registradores_banco[Reg_leitura1]; // LEITURA DOS REGISTRADORES
```

```
25 assign Dados_leitura2 = registradores_banco[Reg_leitura2];
26
27 always @ (negedge clk) // ESCRITA NOS REGISTRADORES
28 begin
29         if(WriteHILO)
30                 begin
31                 registradores_banco[HI] <= ResultadoHILO[63:32];
32                 registradores_banco[LO] <= ResultadoHILO[31:0];
33                 end
34         if(EscreveReg) registradores_banco[Reg_escrita] = Dados_escrita;
35         if (Out) RS = Dados_escrita;
36         if (In) registradores_banco[RE] = Dados_entrada;
37 end
38 endmodule
```

As memórias possuem um funcionamento mais simples do que a maioria dos outros módulos. A memória de dados é uma memória de leitura e escrita, e opera da seguinte forma:

Quando o sinal de controle "EscreveMem"está ativo, a posição da memória apontada pela entrada "endereço"é escrita com os dados da entrada "Dados_escrita". Quando o sinal de controle "LeMem"está ativo, a saída "Dados_leitura"é escrita com os dados da posição de memória apontada pela entrada "endereço".

```
1  module mem_Dados (clk, endereco, Dados_escrita, Dados_leitura, EscreveMem, LeMem);
2
3  input clk, EscreveMem, LeMem;
4  input [31:0] Dados_escrita;
5  input [31:0] endereco;
6  output reg [31:0] Dados_leitura;
7
8  reg [31:0] memoria [255:0];
9
10 always @ (posedge clk)
11 begin
12
13         if (EscreveMem) memoria[endereco] <= Dados_escrita;
14         if (LeMem) Dados_leitura <= memoria[endereco];
15
16 end
17
18 endmodule
```

A memória de instruções é somente de leitura. Seus dados são escritos manualmente na implementação. Os projetos relacionados as próximas disciplinas do curso de Engenharia da Computação implementarão novos modos de trabalhar os dados dessa memória. Na boda de subida do *clock*, a saída "instrucao"recebe os dados da posição de memória apontada pela entrada "endereco_leitura".

```
1  module mem_Instrucao(endereco_leitura, clk, instrucao);
2
3          input [31:0]endereco_leitura;
4          input clk;
5          integer programa = 1;
6
```

```
7         reg [31:0]mem_instrucao[255:0];
8
9         output [31:0] instrucao;
10
11        always @(posedge clk)
12        begin
13                if(programa == 1)
14                        begin
15 ////////////////////INSTRUCOES DO PROGRAMA
       ////////////////////////////////////////////
16                        //mem_instrucao[100] = 32'b00011100011100011100011100011100; //
                              TESTE SIMULA AO
17 //
       ////////////////////////////////////////////////////////////////////////////////////

18                        end
19         programa = 0;
20        end
21        assign instrucao = mem_instrucao[endereco_leitura];
22
23 endmodule
```

Todos os multiplexadores funcionam de modo igual, e servem para fazer com que os sinais de controle executem suas funções, que foram descritas anteriormente. Note que a maior diferença em seus códigos são os sinais de controle utilizados.

```
1  module muxa (Instrucao0_25, Dados_leitura1, JumpR, saida_muxa);
2
3         input [25:0] Instrucao0_25;
4         input [31:0] Dados_leitura1;
5         input JumpR;
6
7         output reg [25:0] saida_muxa;
8
9         always @ (*)
10        begin
11
12                if (JumpR) saida_muxa = Dados_leitura1[25:0];
13                else saida_muxa = Instrucao0_25;
14
15        end
16
17
18
19 endmodule
```

```
1  module muxb (Instrucao21_25, JumpR, saida_muxb);
2
3         input [4:0] Instrucao21_25;
4         parameter ra = 5'b11111;
5         input JumpR;
6
7         output reg [25:0] saida_muxb;
8
9         always @ (*)
10        begin
11
12                if (JumpR) saida_muxb = ra;
13                else saida_muxb = Instrucao21_25;
```

```
14
15          end
16
17
18
19  endmodule
```

```
1   module muxc (Instrucao16_20, Instrucao11_15, RegDst, saida_muxc);
2
3           parameter ra = 5'b11111; // ENDERECO DO REGISTRADOR DE ENDERECO: 11111
4           input [4:0] Instrucao16_20;
5           input [4:0] Instrucao11_15;
6           input [1:0] RegDst;
7
8           output reg [4:0] saida_muxc;
9
10          always @ (*)
11          begin
12
13                  if (RegDst == 2'b00) saida_muxc = Instrucao16_20;
14                  if (RegDst == 2'b01) saida_muxc = Instrucao11_15;
15                  if (RegDst == 2'b10) saida_muxc = ra;
16          end
17
18
19  endmodule
```

```
1   module muxd (PC, Dados_escrita, WriteRa, saida_muxd);
2
3   input [31:0] PC;
4   input [31:0] Dados_escrita;
5   input WriteRa;
6
7   output reg [31:0] saida_muxd;
8
9   always @(*)
10  begin
11
12          if (WriteRa) saida_muxd = PC;
13          else saida_muxd = Dados_escrita;
14
15  end
16
17  endmodule
```

```
1   module muxe (OrigALU, Dados_leitura2, imediato, saida_muxe);
2
3   input OrigALU;
4   input [31:0] Dados_leitura2;
5   input [31:0] imediato;
6
7   output reg [31:0] saida_muxe;
8
9   always @(*)
10  begin
11          if (OrigALU) saida_muxe = imediato;
12          else saida_muxe = Dados_leitura2;
13
14  end
15
```

```
endmodule
```

```
module muxf (entradaOR, PCplus4, PCplusBranch, saida_muxf );

input [31:0] PCplus4;
input [31:0] PCplusBranch;
input entradaOR;

output reg [31:0] saida_muxf;

always @(*)
begin
        if (entradaOR) saida_muxf = PCplusBranch;
        else saida_muxf = PCplus4;
end


endmodule
```

```
module muxg (entrada_muxf, entrada_jump, Jump, saida_muxg);

        input [31:0]entrada_muxf;
        input [31:0]entrada_jump;
        input Jump;

        output reg [31:0]saida_muxg;

        always @(*)
        begin
                if (Jump) saida_muxg = entrada_jump;
                else saida_muxg = entrada_muxf;

        end



endmodule
```

```
module muxk (Dados_leitura, Slt_resultado, Resultado_ula, Slt, MemparaReg, saida_muxk);

        input [31:0] Dados_leitura;
        input Slt_resultado, Slt, MemparaReg;
        input [31:0] Resultado_ula;

        output reg [31:0] saida_muxk;

        always @(*)
        begin
                if (Slt) saida_muxk = Slt_resultado;
                else if (MemparaReg) saida_muxk = Dados_leitura;
                else saida_muxk = Resultado_ula;

        end


endmodule
```

Os módulos de entrada e saída operam de modo semelhante ao PC:

```
module entrada (switches_fpga, In, saida_RE);
```

```
2
3  input [9:0] switches_fpga;
4  input In;
5
6  output reg [31:0] saida_RE;
7
8  always @(*)
9  begin
10 if(In) saida_RE = switches_fpga;
11 end
12
13 endmodule
```

```
1  module saida (out, Dados_saida, saida_display);
2
3  input out;
4  input [31:0] Dados_saida;
5
6  output reg [31:0] saida_display;
7
8  always @(*)
9  begin
10         if(out) saida_display = Dados_saida;
11
12
13 end
14
15 endmodule
```

O módulo "concatenador"possui como função concatenar os 6 dígitos mais significativos do endereço armazenado pelo PC com os 26 bits da instrução do tipo J.

```
1  module concatenador (entrada_AddPC, entrada_jump, saida_endJump);
2          input [31:0] entrada_AddPC;
3          input [25:0] entrada_jump;
4
5          output wire [31:0] saida_endJump;
6          assign saida_endJump = { entrada_AddPC [31:26] , entrada_jump };
7
8  endmodule
```

O módulo "extensor_sinal"possui como função estender o dado de 16 bits de uma instrução do tipo I para um dado de 32 bits.

```
1  module extensor_sinal (sin_entrada, sin_saida);
2
3  input [15:0] sin_entrada;
4
5  output wire [31:0] sin_saida;
6  reg [15:0] extend = 16'b0000000000000000;
7
8  assign sin_saida = {extend, sin_entrada};
9
10 endmodule
```

5 Resultados obtidos e Discussões

Nesta etapa do projeto, o único modo de testar o funcionamento dos módulos implementados é utilizar a simulação em forma de onda, do *software* utilizado. Os blocos mais complexos e fundamentais foram testados de forma mais meticulosa, enquanto os módulos mais simples foram testados de modo a apenas confirmar seu funcionamento.

5.1 Simulação da Unidade de Lógica e Aritmética

Para realizar os teste com a Unidade de Lógica e Aritmética, as entradas que correspondem aos operandos e ao deslocamento foram fixadas e a entrada de controle, um barramento de 4 bits que define a operação a ser realizada, foi testada em todas as possibilidades, de modo que todas as operações que podem ser realizadas pela ULA foram testadas. As figuras 3 e 4 ilustram os 2 testes realizado com esse módulo. No primeiro, os operandos foram dois números positivos, enquanto que, no segundo, um foi positivo e um negativo. Vale ressaltar que é possível identificar as operações realizadas nestas primeiras duas simulações através da tabela 1.

Note que, como uma das decisões de projeto do processador teve como ideia fazer com que a saída da ULA possua 64 bits, para haver a possibilidade de escrever os registradores HI e LO, no banco de registradores, ao mesmo tempo, de modo que os 32 bits mais significativos do resultado sejam escritos no registrador HI, e os 32 menos significativos no registrador LO, a operação de divisão apresenta um resultado, a primeira vista, estranho. Mas o modo com que a divisão foi definida, onde HI recebe o resto da divisão e LO o quociente, faz com que a saída "Resultado"receba esse número, em decimal.

Vale ressaltar que as operações lógicas são realizadas bit a bit, e que, a partir do controle "1011", o resultado não é relevante, uma vez que não existem operações para os próximos sinais.

Figura 3 – Primeira simulação em forma de onda da Unidade de Lógica e Aritmética.

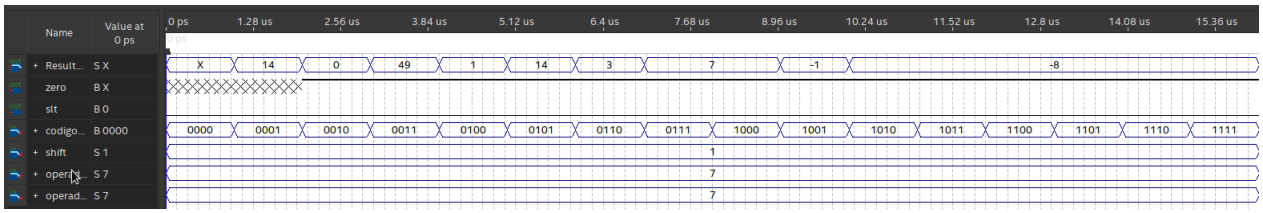
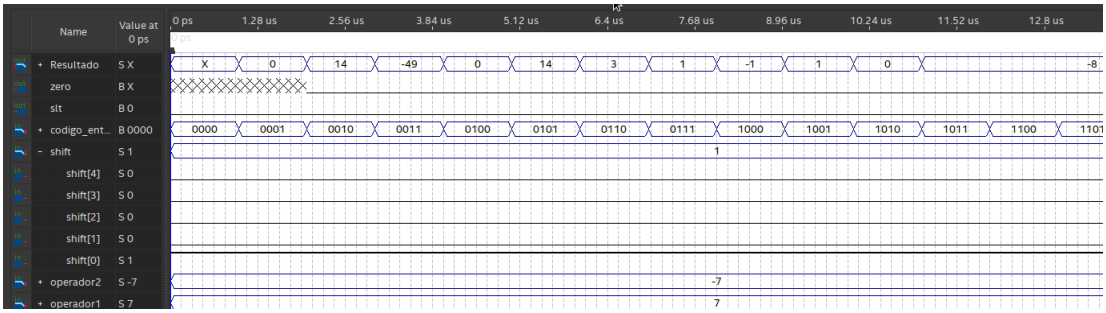


Figura 4 – Segunda simulação em forma de onda da Unidade de Lógica e Aritmética.



5.2 Simulação do Banco de Registradores

O banco de registradores, por ser um circuito que depende de um *clock*, foi testado de modo diferente dos demais. Tendo em vista que foi projetado para realizar as escritas na descida de *clock* e as leituras na subida, a simulação procedeu da seguinte forma :

- Na primeira e na segunda descida de *clock*, como se pode observar na figura 5, foi realizado a escrita do número 7 e do número 5 respectivamente, no banco de registradores, nos endereços 1 e 2, como se pode observar na simulação.

- Na primeira subida de *clock* após a escrita, os 2 registradores são lidos e, como se pode observar, aparecem nas saídas, assim como o esperado.

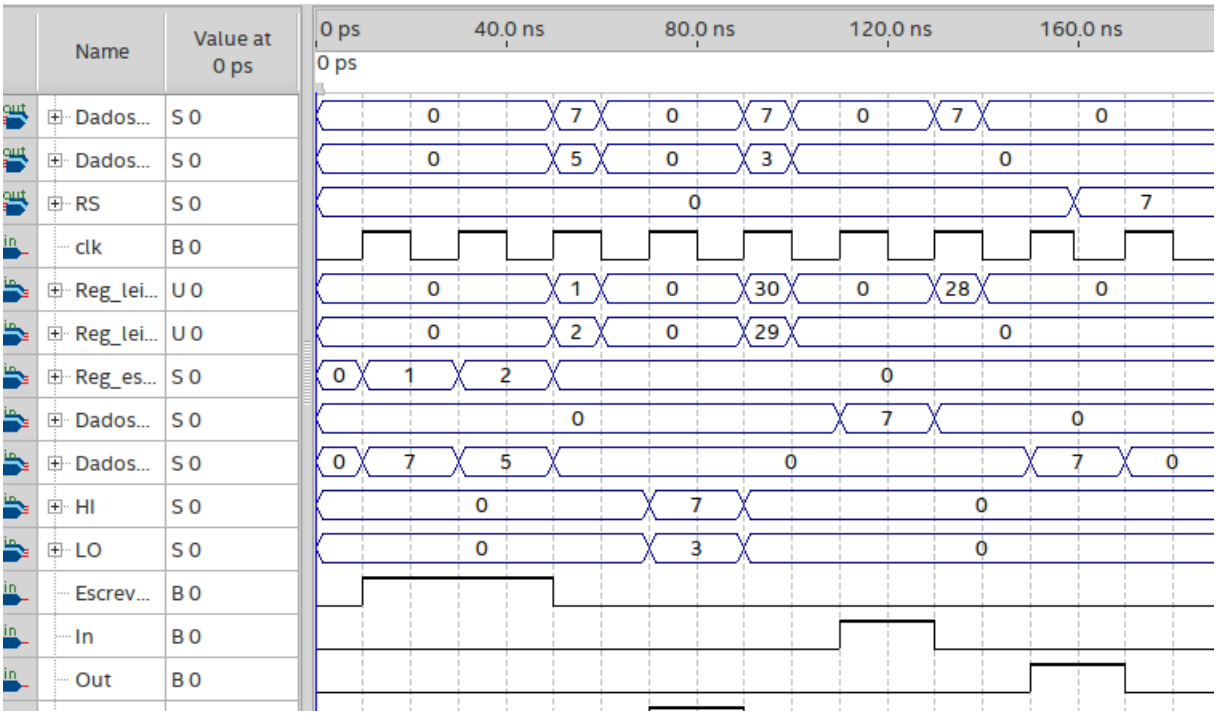
- Na próxima descida de *clock*, correponde ao momento em que o tempo é 80 ns, uma escrita nos registradores HI e LO é realizada. A entrada do sinal de controle "WriteHILO"acabou sendo cortada da imagem, mas ela fica alta entre os momentos correspondentes ao tempo em 70 ns a 90 ns. Os números escritos foram os mesmos da primeira escrita.

- Após isso, uma nova leitura é realizada no banco de registradores, desta vez, como se pode observar, nos registradores HI e LO, que estão no endereço 29 e 30, respectivamente, como se pode notar na imagem.

- Por último, na próxima descida de *clock*, o registrador de entrada, que possui o endereço 28 é escrito com o número 7, e na próxima subida de *clock* é realizada uma leitura neste registrador, que retorna o número 7.

Sendo assim, todas as funções do banco de registradores foram devidamente testadas e o módulo apresenta um funcionamento coerente.

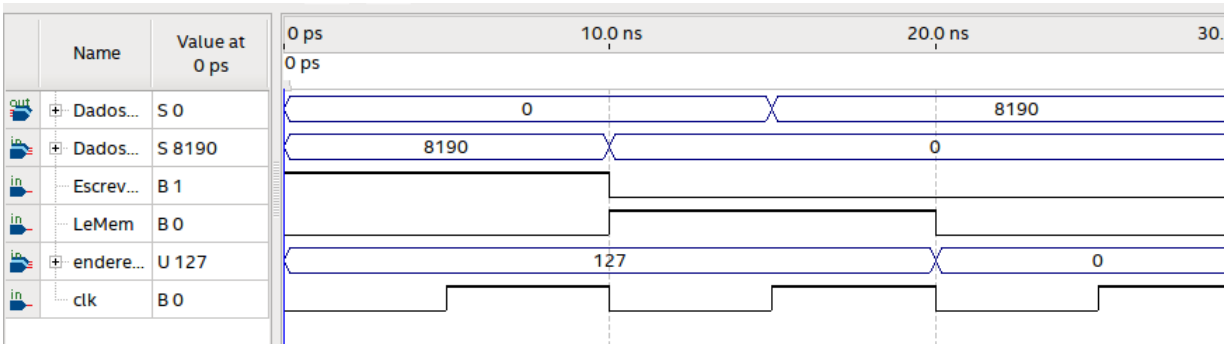
Figura 5 – Simulação em forma de onda realizada no banco de registradores.



5.3 Simulação da Memória de Dados

As memórias apresentam uma complexidade menor. Na memória de dados, como se pode observar na figura 6, foi realizado apenas uma escrita e uma leitura, no mesmo endereço, e tudo funcionou corretamente.

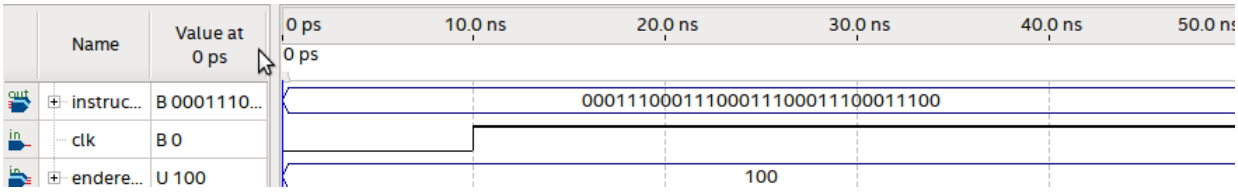
Figura 6 – Simulação em forma de onda da Memória de Dados.



5.4 Simulação da Memória de Instruções

Como a memória de instruções é uma memória somente de leitura, houve a implementação prévia de um valor no endereço 100, como se pode observar no código desse módulo no capítulo anterior, e a simulação de uma leitura nesse mesmo endereço. Tudo funcionou corretamente.

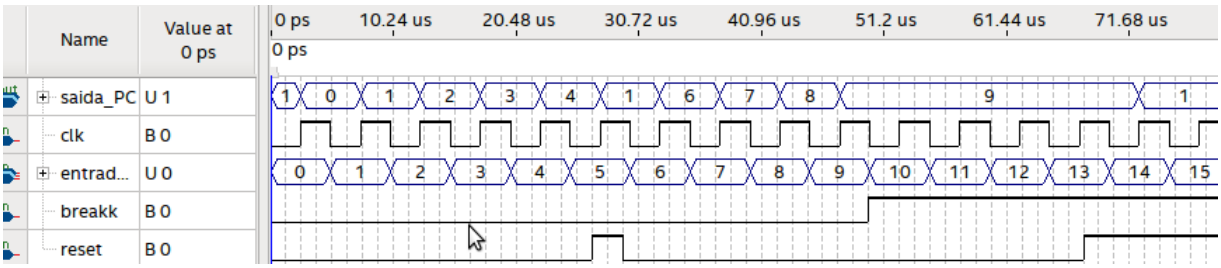
Figura 7 – Simulação em forma de onda realizada na Memória de Instruções.



5.5 Simulação do Contador de Programa

Como dito anteriormente, o Contador de Programa, bloco denotado por "PC"se comporta basicamente como um registrador, do mesmo modo que os módulos de entrada e saída. Sendo assim, uma vez que a simulação do "PC"confirmar seu funcionamento, não é necessário simular esses últimos, uma vez que seu funcionamento é equivalente Note que todas as funções apresentada do contador de programa são testadas e apresentam um funcionamento coerente.

Figura 8 – Simulação em forma de onda realizada no PC.

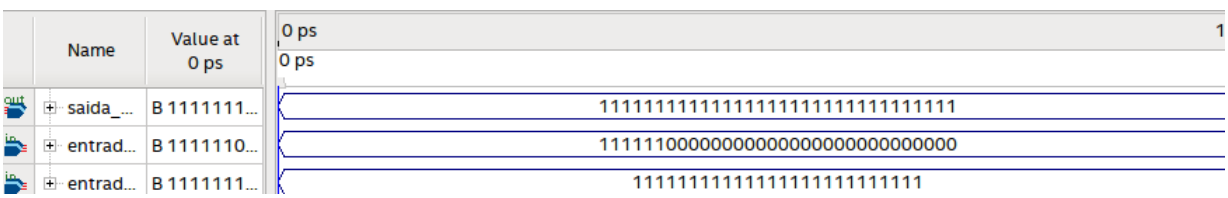


5.6 Simulação dos multiplexadores, do concatenador e do extensor

Pelo fato dos multiplexadores possuírem um funcionamento equivalente o simulado foi o mais particular entre eles. Quando o sinal de controle "Slt"está ativo, o módulo retorna a entrada "Slt_resultado". Quando ambos os sinais de controle estão desativados, o módulo retorna a entrada "Resultado_ula". Quando o sinal "MemparaReg"está ativo, o módulo retorna a entrada "Dados_leitura".

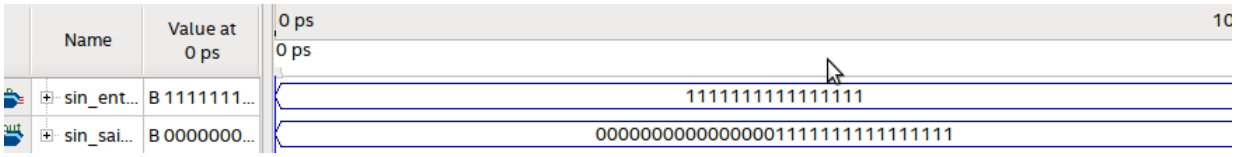
Para testar o concatenador as entradas foram preenchidas com números arbitrários e a simulação confirmou que, de fato, a concatenação foi realizada.

Figura 9 – Simulação em forma de onda realizada no PC.



No extensor de sinal, um procedimento semelhante ao concatenador foi realizado. Uma entrada arbitrária foi gerada, e a simulação confirma que a extensão de sinal de fato ocorre.

Figura 10 – I



6 Considerações Finais

A principal dificuldade encontrada nesta etapa do projeto foi com a manipulação do *software* utilizado, que apresenta vários *bugs* e inconsistências, tanto no momento de compilação como no momento de simulação. Mas, com algumas adaptações para contornar esses problemas, foi possível concluir a proposta do Ponto de Checagem 2 de modo satisfatório.

Depois do primeiro Ponto de Checagem, foi estudado um modo de interação do processador com dispositivos externos da placa utilizada. E assim, os módulos de entrada e de saída, assim como seus respectivos sinais de controle e instruções de operação foram adicionados ao projeto. Posteriormente, houveram algumas correções de inconsistências do caminho de dados e uma nova versão foi elaborada.

O próximo passo foi planejar meticulosamente como cada módulo iria operar e quais seriam as sua funções, codificando e elaborando os modos com que cada um poderia operar suas entradas. Depois, houve a implementação desses módulos em Verilog, utilizando o *software* Quartus II.

As próximas etapas envolvem a implementação da unidade de controle do processador e o posterior agrupamento de todos os blocos implementados para gerar o produto final.

Referências

1 TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. *Sistemas Digitais: Princípios e Aplicações*. 11th edition. ed. São Paulo: Pearson Education, 2011. Citado na página 9.

2 TERASIC TECHNOLOGIES. *DE2-115 User Manual*. Hsinchu, Taiwan, 2013. Citado na página 11.

3 HENNESSY, J. L.; PATTERSON, D. A. *Arquitetura de Computadores: Uma Abordagem Quantitativa*. 3th edition. ed. Rio de Janeiro, RJ: Campus, 2003. Citado na página 13.

4 PATTERSON, D. A.; HENNESSY, J. L. *Organização e Projeto de Computadores: a interface hardware/software*. 5th edition. ed. Rio de Janeiro, RJ: Campus, 2005. Citado na página 13.

5 CRAIBAS, J. J. S. *Dicas de implementação de circuitos digitais em verilog através do software Quartus Prime*. São Paulo. Apostila disponibilizada pelo docente Prof. Dr. Tiago de Oliveira, na disciplina Laboratório de Arquitetura e Organização de Computadores. Citado na página 15.