Gabriel Kenji de Almeida

Implementação de um processador monociclo para o FPGA com o Quartus Prime.

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 de um processador capaz de executar uma variedade de instruções possibilitem a execução dos algoritmos mais básicos da literatura de Lógica de Programação. O projeto final será simulado no 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 elaboração do projeto se deu pela realização gradativa durante o semestre de 4 etapas principais. Primeiramente, houve a escolha de particularidades e características do projeto, e a construção de um caminho de dados. Posteriormente, houve a elaboração e a implementação das unidades de processamento e de controle, de modo que o processador possua a capacidade de executar todas as instruções propostas inicialmente. E, por fim, houve a realização de testes e simulações com o algoritmo implementado.

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

Lista de ilustrações

Lista de tabelas

Sumário

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 processadores.

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 primeira etapa do projeto, como foi definido no Ponto de Checagem 1, teve como finalidade a escolha do conjunto de instruções e da arquitetura base do projeto, assim como todos os aspectos técnicos e conceituais a serem trabalhados no desenvolvimento do processador. Sendo assim, pontuando cada aspecto a trabalhado, tem-se:

- Escolha de um conjunto de instruções apropriado para executar algoritmos básicos.
- Definição de um conjunto de formatos de instrução que comporte todas as instruções escolhidas.
- Escolha dos modos de endereçamento com os quais o processador irá operar.
- Escolha de um conjunto de registradores a serem utilizados pelo processador durante as operações.
- Elaboração de um diagrama que agrupe todos os circuitos lógicos utilizados em blocos para ilustrar cada módulo utilizado no processador e como eles interagem entre si.
- Após a elaboração do diagrama, apresentar os sinais de controle escolhidos para a elaboração da unidade de controle do processador.

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.

Por último a terceira e quarta etapa, que contemplaram os Pontos de Checagem 3 e 4, tiveram por finalidade e elaboração da Unidade de Controle, a interligação entre todos os módulos desenvolvidos, e a realização de testes no FPGA com o protótipo obtido.

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, conceito que será esclarecido posteriormente, 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 Formatos de instruções

No caso do processador a ser implementado, uma instrução conterá 32 bits. Um formato de instrução, de forma resumida, seria o modo com que essa sequência de bits será interpretada pelo processador. Geralmente, instruções que realizam operações semelhantes possuem o mesmo formato de instrução. No caso do projeto desenvolvido, foi decido que haverá o uso de 3 formatos de instruções diferentes, do mesmo modo que ocorre no modelo MIPS. Esses formatos e como eles funcionam serão discutidos no próximo capítulo.

3.4 Modos de Endereçamento

O modo de endereçamento denota o modo com que o processador acessa a memória, a partir da instrução executada no momento. Uma escolha de modos de endereçamento apropriada permite o maior uso de memória, como será perceptível no próximo capítulo, quando esse assunto for discutido.

3.5 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.

3.6 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.

3.7 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 possibilidade de uso de todas as operações lógicas e aritméticas.(5)

4 Desenvolvimento

É importante agrupar nesta seção, todas as características da arquitetura base deste projeto que foram citadas e explicadas no capítulo anterior. Sendo assim, a arquitetura possuirá um tamanho de instruções de 32 bits e uma memória mapeada em palavras de 32 bits. O processador será monociclo, ou seja, realizará uma instrução por ciclo de clock.

4.1 Conjunto de instruções escolhido e suas particularidades

Lembrando que os conjunto de instruções adotado segue as características de um conjunto RISC, que ja foi conceituado anteriormente. Para expor o conjunto adotado, haverá a sitação de cada instrução escolhida junto com uma breve descrição de sua utilizade. Vale ressaltar que o processador desenvolvido utilizará o complemento de 2 para representar números com sinal e do modo little endian, donde os números são apresentados com o digito mais significativo a esquerda e o menos significativo a direita.

Instruções de Operações Aritméticas

- **add**: Soma básica entre 2 valores armazenados em registradores.
- **addi**: Soma básica entre um valor armazenado em um registrador e um imediato. Note que um imediato é um número represetado diretamente no código da instrução.
- **sub**: Subtração básica de dois valores armazenados em registradores.
- **subi**: Substração básica de um valor armazenado em um registrador com um imediato.
- **mult**: Multiplicação de dois valores armazenados em registradores. Os 32 bits mais significativos do resultados são armazenados no registrador HI e os 32 menos significativos nos registrador LO.
- **div**: Divisão inteira entre dois valores armazenados em registradores. O resultado da divisão é armazenado no registrador LO e o resto no HI.

Instruções de Transferência de Informação:

- **lw**: Carrega uma palavra de 32 bits da memória para um registrador do banco de registradores.
- **sw**: Salva uma palavra de 32 bits armazenada em um registrador na memória.

Instruções de Deslocamento de bits:

- **sra**: Desloca de modo aritmético um número para a direita.
- **sla**: Desloca de modo aritmético um número para a esquerda.

Vale ressaltar que o deslocamento aritmético, diferente do deslocamento lógico, sempre multiplica por 2 (deslocando para a direita) ou divide por 2 (deslocando para a esquerda) um número binário, mesmo quando o número é representado em complemento de 2. No caso do deslocamento lógico, isso só funciona para números sem sinal.

Instruções de Operações Lógicas e de Comparação:

- **and**: Realiza uma operação de and lógica entre valores armazenados em 2 registradores.
- **andi**: Realiza uma operação de and lógica entre um valor imediato e um valor armazenado em um registrador.
- **or**: Realiza uma operação de or lógica entre valores armazenados em 2 registradores.
- **ori**: Realiza uma operação de or lógica entre um valor imediato e um valor armazenado em um registrador.
- **xor**: Realiza uma operação xor entre valores armazenados em 2 registradores.
- **xori**: Realiza uma operação xor com um valor armazenado em um registrador e um imediato.
- **nor**: Realiza uma operação nor lógica entre valores armazenados em 2 registradores.
- **slt**: Compara o valor de 2 registradores através de uma subtração. Se o valor do primeiro registrador for menor do que o do segundo, um regitrador especificado pelo código da instrução recebe o valor 1.
- **slti**: Compara o valor de um registrador com um imediato. Se o valor do registrador for menor do que o imediato, um registrador especificado pelo código da instrução recebe o valor 1.
- **not**: Realiza uma operação de negação com o valor de um registrador.

Instruções de Salto e desvio:

- **j**: Operação que realiza um salto do endereço da instrução atual para um outro endereço da memória de instruções, determinado pelo código da instrução. Desse modo, há a possibilidade de "pular"um uma determinada quantidade de instruções, que acabam deixando de serem executadas. O modo como isso ocorre será explicado quando o diagrama em blocos do processador for apresentado.

- **jr**: Operação que realiza um salto do endereço da instrução atual para o endereço armazenado no registrador $ra.

- **beq**: Compara o valor de 2 registradores. Se forem iguais, um desvio é realizado para o endereço especificado no código da instrução. Um desvio pode ser entendido como um salto que ocorre condicionalmente.

- **bne**: Compara o valor de 2 registradores e realiza um desvio para um endereço especificado pelo código da instrução se esses dois valores forem diferentes.

- **jal**: Realiza a mesma operação da instrução j, porém salva o valor do endereço atual no registrador $ra. Essa instrução é utilizada em conjunto com a instrução jr para possibilitar a criação de funções na memória de instruções.

Vale ressaltar que, caso necessário, existe a possibilidade da inclusão de novas instruções no conjunto trabalhado, do mesmo modo que há a possibilidade da exclusão de instruções já existentes.

4.2 Formatos de instruções escolhidos

Os formatos de instruções escolhidos neste projeto são os mesmos dos utilizados pelo MIPS. As tabelas 1, 2 e 3 apresentam como esses três formatos escolhidos estão organizados. Vale ressaltar que o modo com que as tabelas apresentam os campos e a respectiva quantidade de bits reservada a esses campos também segue a ordem little endian. Desse modo, os 6 bits mais significativos da instrução representam o Opcode, que aparece mais à direita na tabela, e assim por diante. É importante ressaltar também a função de cada um dos campos utilizados para rotular cada parcela do código de instrução.

- Os campos **Opcode** e **funct** denotam os bits utilizados pelo processador com a função de identificar a instrução a ser executada.

- Os campos **R1** e **R2** armazenam os endereços para os registradores a serem acessados no banco de registradores que possuem os operandos da instrução.

- O campo **Rd** denota o endereço do registrador de destino, o registrador do banco de registradores que receberá o resultado da operação executada pelo processador.

- O campo **shamt** É utilizado exclusivamente para as instruções de deslocamento e armazena o número de deslocamentos a ser realizado pela instrução.
- O campo **Imediato**, exclusivo para instruções do tipo I, é utilizado para armazenar um número ou endereço diretamente no código da instrução, que será utilizado de modo direto ou imediato na execução da instrução.
- O campo **Endereço**, exclusivo para instruções do tipo J, é utilizado para armazenar diretamente no código da instrução, o endereço de destino do salto a ser realizado.

Tabela 1 – Instruções do tipo R

Opcode	R1	R2	Rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Tabela 2 – Instruções do tipo I

Opcode	R1	Rd	Imediato
6 bits	5 bits	5 bits	16 bits

Tabela 3 – Instruções do tipo J

Opcode	Endereço
6 bits	26 bits

Um dos aspectos sobre o conjunto de instruções que foi omitido neste relatório fo

4.3 Modos de endereçamento escolhidos

Como já dito no capítulo anterios, os modos de endereçamentos são os modos com os quais o processador utiliza o código de instrução para acessar os dados necessários nas operações.

Os modos de endereçamento escolhidos, baseados no modelo MIPS, foram 5:

- **Imediato**, onde o operando é uma constante. Será utilizado para instruções do tipo I de operações lógicas e aritméticas. Uma instrução que utilizará esse método de endereçamento é o **addi**.
- **Por Registrador**, onde as constantes da instrução carregam um endereço para um registrador. Será utilizado para todas as instruções do tipo R, como por exemplo, **add**.
- **Por deslocamento**, onde o operando é um endereço de memória estabelecido a partir da soma de um registrador e uma constante da instrução. Será utilizado para instruções de transferência de informação do tipo I, como por exemplo, **lw**.

- **Relativo a PC**, usado para saltos do tipo I e do tipo J, é semelhante ao endereçamento por deslocamento, mas o registrador utilizado é o PC. O PC é um registrador que possui uma função especial, que será descrito na seção de registradores utilizados. Um exemplo de instrução que usa esse modo de endereçamento é o **beq**.

- **Pseudo Direto**, usado para instruções do tipo J, o endereço do salto é deslocado 2 bits à esquerda e concatenado com os 4 bits mais significativos do PC. Esse é o modo com o qual o endereço de um salto é calculado. Instruções que utilizam esse modo são **j, jr e jal**.

Após a descrição da escolha de todos esses elementos para o desenvolvimento do processador, será possível visualizar, no diagrama em bloco a ser apresentado em alguma das próximas seções, como que cada um desses elementos compõem o processador.

4.4 Sinais de controle utilizados

Como já explicado na seção anterior, os sinais de controle possuem o propósito de controlar o funcionamento de cada módulo do processador, de acordo com a instrução executada. No caso do projeto desenvolvido, os sinais de controle escolhidos diferem do MIPS em alguns aspectos, com a ideia de possibilitar a implementação de algumas instruções adicionais. Vale ressaltar novamente que eles serão melhor compreendidos após a exibição do diagrama em bloco do processador. Os sinais de controle escolhidos foram:

- **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 reproduzido no conjunto de *displays* de 7 segmentos;
- **break**: Este sinal, que recebeu o mesmo nome da instrução, quando ativo, faz com que o processador não realize mais nenhuma instrução.

4.5 Conjunto de registradores escolhidos

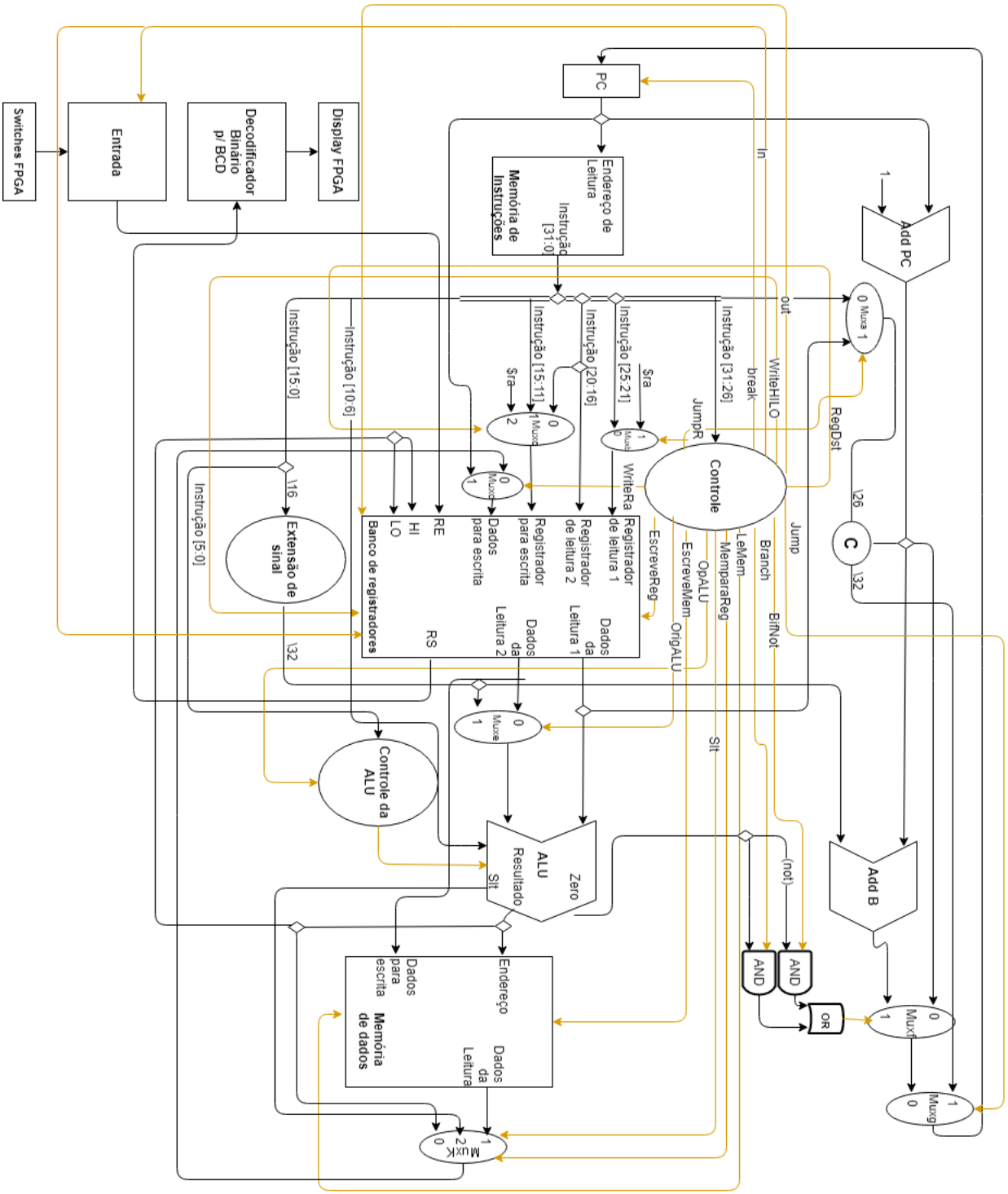
Para este projeto, optou-se por utilizar 34 registradores, com funções e particularidades pré definidas, sendo que 33 deles estarão presentes no módulo denotado por Banco de Registradores. Entre os registradores utilizados, tem-se:

- 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 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.

4.6 Diagrama em bloco do processador

A seguir, tem-se o caminho de dados ou diagrama em bloco do processador a ser projetado a partir de todas as escolhas e decisões apresentadas nas seções anteriores:

Figura 1 – Última versão do caminho de dados, desenhado através do uso da ferramenta do site <draw.io>



O diagrama foi desenvolvido no menor espaço possível, para possibilitar a visualização neste relatório, mas mesmo assim, a ilustração acabou por ocupar muito espaço. Como o caminho de dados ficou bem denso, haverá uma breve explicação para facilitar o entendimento. Uma particularidade desta ilustração é o uso de losangos pequenos em cima dos fios para indicar uma bifurcação, quando é necessário enviar a mesma informação para dois ou mais módulos distintos. As setas coloridas representam os sinais de controle.

Vale ressaltar que, como a memória vai ser mapeada em palavras de 32 bits, diferente do que ocorre no MIPS, a cada instrução o PC será incrementado de modo unitário.

As instruções do formato **J** ocorrem da seguinte forma: Primeiramente, a Memória de Instruções recebe do registrador PC o Endereço de leitura da instrução a ser executada e a envia para um barramento indicado pelos traços duplos, que contém a instrução toda. Logo depois disso, os 26 bits menos significativos são enviados ao único multiplexador vertical da ilustração, que possui a função de distinguir se a operação de salto se trata de uma instrução "j"ou "jr". Logo após isso, o sinal é concatenado com os 6 bits mais significativos do PC, para formar o endereço do desvio, e, assim, o valor obtido é enviado para o PC.

Para explicar como ocorre uma instrução do tipo **I**, vamos utilizar a instrução "beq". Assim como nas instruções do formato anterior e nas demais, em um primeiro momento, a memória de instruções passa para o barramento denotado pelas barras duplas a instrução a ser executada. No caso de uma instrução "beq", o banco de registradores recebe o endereço dos 2 registradores a serem comparados envia seus respectivos valores para a ALU. Enquanto isso, O campo "Imediato"da instrução é enviado para a unidade de "Extensão de sinal"na parte inferior do diagrama, que possui a finalidade de estender o sinal de 16 bits para 32, afim de possibilitar a soma com o endereço atual do PC, e enviar o resultado para o PC novamente, caso o desvio seja tomado. Em um terceiro momento, a ALU realiza a subtração dos operandos que recebeu e, caso a subtração resulte em 0, significando que os valores são iguais, o sinal de saída "Zero"da ALU é enviado para região superior direita da ilustração, que possui uma lógica que faz com que o desvio seje tomado caso o sinal de saída apropriado seja enviado.

Uma instrução do tipo **R** ocorre da seguinte forma: Após a leitura e a passagem da instrução para o barramento de barras duplas, a unidade de controle faz com que os campos "Registrador de leitura 1"receba o campo "R1"da instrução, "Registrador de leitura 2"receba o campo "R2"da instrução e o campo "Registrador para escrita"receba o campo "Rd"da instrução. Posteriormente, os valores de "R1"e "R2"são enviados para a ALU, que realiza a operação e envia o resultado para o campo "Dados para escrita"que acabam por serem escritos no registrador apontado pelo campo "Rd", presente em "Registrador para escrita"

4.7 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 4 – 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 procedimentos. 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".

```
module mem_Dados (clk, endereco, Dados_escrita, Dados_leitura, EscreveMem, LeMem);

input clk, EscreveMem, LeMem;
input [31:0] Dados_escrita;
input [31:0] endereco;
output reg [31:0] Dados_leitura;

reg [31:0] memoria [255:0];

always @ (posedge clk)
begin

        if (EscreveMem) memoria[endereco] <= Dados_escrita;
        if (LeMem) Dados_leitura <= memoria[endereco];

end

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 "instrução"recebe os dados da posição de memória apontada pela entrada "endereço_leitura".

```
module mem_Instrucao(endereco_leitura, clk, instrucao);

        input [31:0]endereco_leitura;
        input clk;
        integer programa = 1;

        reg [31:0]mem_instrucao[255:0];

        output [31:0] instrucao;

        always @(posedge clk)
        begin
                if(programa == 1)
                        begin
//////////////////////INSTRUCOES DO PROGRAMA
    ////////////////////////////////////////////
                        //mem_instrucao[100] = 32'b00011100011100011100011100011100; //
                            TESTE SIMULA AO
//
    ////////////////////////////////////////////////////////////////////////////////////

                        end
         programa = 0;
        end
        assign instrucao = mem_instrucao[endereco_leitura];

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.

```
module muxa (Instrucao0_25, Dados_leitura1, JumpR, saida_muxa);

        input [25:0] Instrucao0_25;
        input [31:0] Dados_leitura1;
        input JumpR;

        output reg [25:0] saida_muxa;

        always @ (*)
        begin

                if (JumpR) saida_muxa = Dados_leitura1[25:0];
                else saida_muxa = Instrucao0_25;

        end



endmodule
```

```
module muxb (Instrucao21_25, JumpR, saida_muxb);

        input [4:0] Instrucao21_25;
        parameter ra = 5'b11111;
        input JumpR;

        output reg [25:0] saida_muxb;

        always @ (*)
        begin

                if (JumpR) saida_muxb = ra;
                else saida_muxb = Instrucao21_25;

        end



endmodule
```

```
module muxc (Instrucao16_20, Instrucao11_15, RegDst, saida_muxc);

        parameter ra = 5'b11111; // ENDERECO DO REGISTRADOR DE ENDERECO: 11111
        input [4:0] Instrucao16_20;
        input [4:0] Instrucao11_15;
        input [1:0] RegDst;

        output reg [4:0] saida_muxc;

        always @ (*)
        begin

                if (RegDst == 2'b00) saida_muxc = Instrucao16_20;
                if (RegDst == 2'b01) saida_muxc = Instrucao11_15;
                if (RegDst == 2'b10) saida_muxc = ra;
        end


endmodule
```

```
module muxd (PC, Dados_escrita, WriteRa, saida_muxd);

input [31:0] PC;
input [31:0] Dados_escrita;
input WriteRa;

output reg [31:0] saida_muxd;

always @(*)
begin

        if (WriteRa) saida_muxd = PC;
        else saida_muxd = Dados_escrita;

end

endmodule
```

```
module muxe (OrigALU, Dados_leitura2, imediato, saida_muxe);

input OrigALU;
input [31:0] Dados_leitura2;
input [31:0] imediato;

output reg [31:0] saida_muxe;

always @(*)
begin
        if (OrigALU) saida_muxe = imediato;
        else saida_muxe = Dados_leitura2;

end

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;

```

```
9         always @(*)
10        begin
11                if (Jump) saida_muxg = entrada_jump;
12                else saida_muxg = entrada_muxf;
13
14        end
15
16
17
18  endmodule
```

```
1   module muxk (Dados_leitura, Slt_resultado, Resultado_ula, Slt, MemparaReg, saida_muxk);
2
3         input [31:0] Dados_leitura;
4         input Slt_resultado, Slt, MemparaReg;
5         input [31:0] Resultado_ula;
6
7         output reg [31:0] saida_muxk;
8
9         always @(*)
10        begin
11                if (Slt) saida_muxk = Slt_resultado;
12                else if (MemparaReg) saida_muxk = Dados_leitura;
13                else saida_muxk = Resultado_ula;
14
15        end
16
17
18  endmodule
```

O módulo de entrada opera de modo semelhante ao PC:

```
1   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
```

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.

```
module extensor_sinal (sin_entrada, sin_saida);

input [15:0] sin_entrada;

output wire [31:0] sin_saida;
reg [15:0] extend = 16'b0000000000000000;

assign sin_saida = {extend, sin_entrada};

endmodule
```

Pelo fato do processador desenvolvido adotar a mesma abordagem de um processador MIPS monociclo, a unidade de controle consiste basicamente em um circuito combinacional decodificador, cuja entrada é o campo "Opcode"da instrução executada, e as saídas são os sinais controle.

Figura 2 – Amostra de código da Unidade de Controle.

```
module unidade_Controle(Opcode, WriteHILO, EscreveReg, OrigALU, LeMem, EscreveMem, MemparaReg,
WriteRa, BifNot, Slt, Jump, JumpR, Out, In, Break, Branch, OpALU, RegDst);

    input [5:0] Opcode;

    output reg WriteHILO, EscreveReg, OrigALU, LeMem, EscreveMem, MemparaReg,
    WriteRa, BifNot, Slt, Jump, JumpR, Out, In, Break, Branch;

    output reg [1:0] OpALU, RegDst;

    always @(*)
    begin

        case(Opcode)

            6'b000001:
                begin
                    WriteHILO = 0;
                    EscreveReg = 1;
                    OrigALU = 0;
                    LeMem = 0;
                    EscreveMem = 0;
                    MemparaReg = 0;
                    WriteRa = 0;
                    BifNot = 0;
                    Slt = 0;
                    Jump = 0;
                    JumpR = 0;
                    Out = 0;
                    In = 0;
                    Break = 0;
                    Branch = 0;
                    OpALU = 2'b00;
                    RegDst = 2'b01;
                end
```

A implementação deste módulo consistiu basicamente em um bloco *"case"* com a atribuição de todos os sinais de controle para cada valor que o campo Opcode pode assumir, de acordo com a funcionalidade da instrução que o Opcode representa, e com as funções de cada sinal de controle, que foram descritas anteriormente. Vale ressaltar que o código foi apresentado em forma de imagem por se tratar de apenas uma amostra.

A interligação entre todos os módulos também possui uma complexidade relativamente simples. Houve a declaração de variáveis do tipo *wire* para representar cada

fio e barramento exposto no caminho de dados. Novamente, por se tratar de um código repetitivo, que consiste basicamente na chamada de todos os blocos elaborados, tem-se apenas um trecho do código:

Figura 3 – Amostra de código da interligação de todos os blocos (Declaração das variáveis).

```
module processador (Clk, Chaves, saida_Display, Reset, SaidaTeste, Resultado_ULA, saidaMemDados);

    input Clk, Reset;
    input [9:0] Chaves;
    output wire [31:0] saida_Display, SaidaTeste, saidaMemDados; //// ULTIMA VARIAVEL PARA TESTE, RETIRAR DEPOIS


    //SINAIS DE CONTROLE/////////////////////////////////////////////////////
    wire WriteHILOwire, EscreveRegwire, OrigALUwire,
    LeMemwire, EscreveMemwire, MemparaRegwire, WriteRawire, BifNotwire, Sltwire,
    Jumpwire, JumpRwire, Outwire, Inwire, Breakwire, Branchwire;

    wire [1:0] OpALUwire, RegDstwire;

    wire[3:0] controleALU;

    ////////////////////////////////////////

    output wire [63:0] Resultado_ULA;

    wire [31:0] Saida_addPC, saida_moduloEntrada, saida_leitura2, saida_RS,
    saida_Concatenador, saida_AddB, saida_MemData, saidaPC, saida_Muxg, saida_Muxf,
    Instrucao, saida_Extensor, saida_Muxe, saida_leitura1, saida_Muxk, saida_Muxd;

    wire [4:0] saida_Muxc;

    wire [25:0] saida_Muxa;

    wire [4:0] saida_Muxb;

    wire slt_ula, zero_ula, saida_branch_logic;
```

Figura 4 – Amostra de código da interligação de todos os blocos (Chamada dos blocos).

```
unidade_Controle UndControle(
        .Opcode(Instrucao[31:26]),
        .WriteHILO(WriteHILOwire),
        .EscreveReg(EscreveRegwire),
        .OrigALU(OrigALUwire),
        .LeMem(LeMemwire),
        .EscreveMem(EscreveMemwire),
        .MemparaReg(MemparaRegwire),
        .WriteRa(WriteRawire),
        .BifNot(BifNotwire),
        .Slt(Sltwire),
        .Jump(Jumpwire),
        .JumpR(JumpRwire),
        .Out(Outwire),
        .In(Inwire),
        .Break(Breakwire),
        .Branch(Branchwire),
        .OpALU(OpALUwire),
        .RegDst(RegDstwire)

);
controle_ALU UndContALU(
        .OpAlu(OpALUwire),
        .Opcode(Instrucao[31:26]),
        .controle_ULA(controleALU)

);


Program_counter pc(
        .breakk(Breakwire),
        .entrada_PC(saida_Muxg),
        .saida_PC(saidaPC),
        .clk(Clk),
        .reset(Reset)

);
```

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

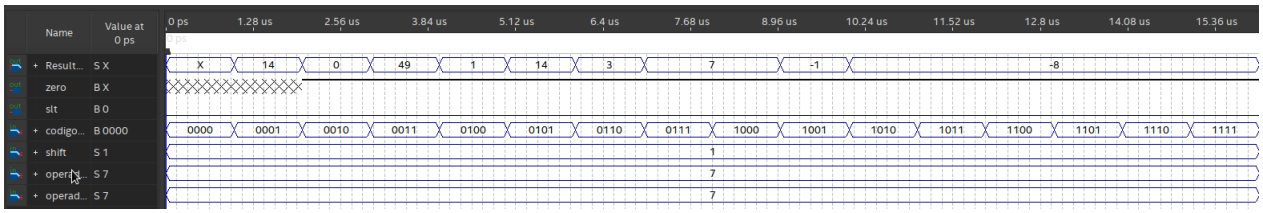
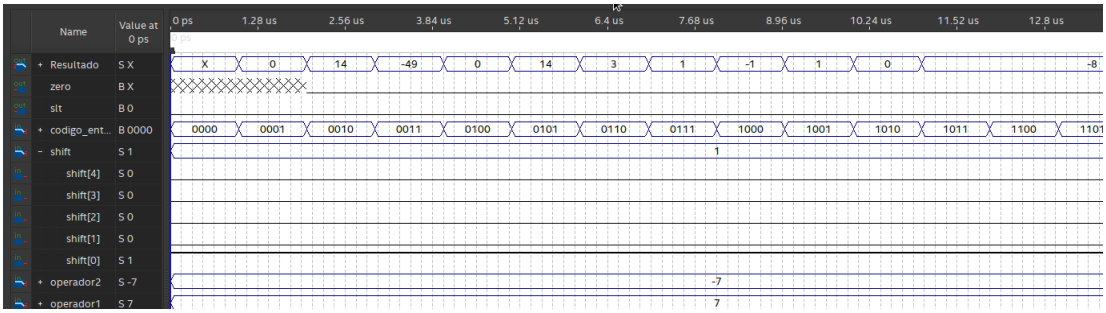


Figura 6 – Segunda simulação em forma de onda da Unidade de Lógica e Aritmética.



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.

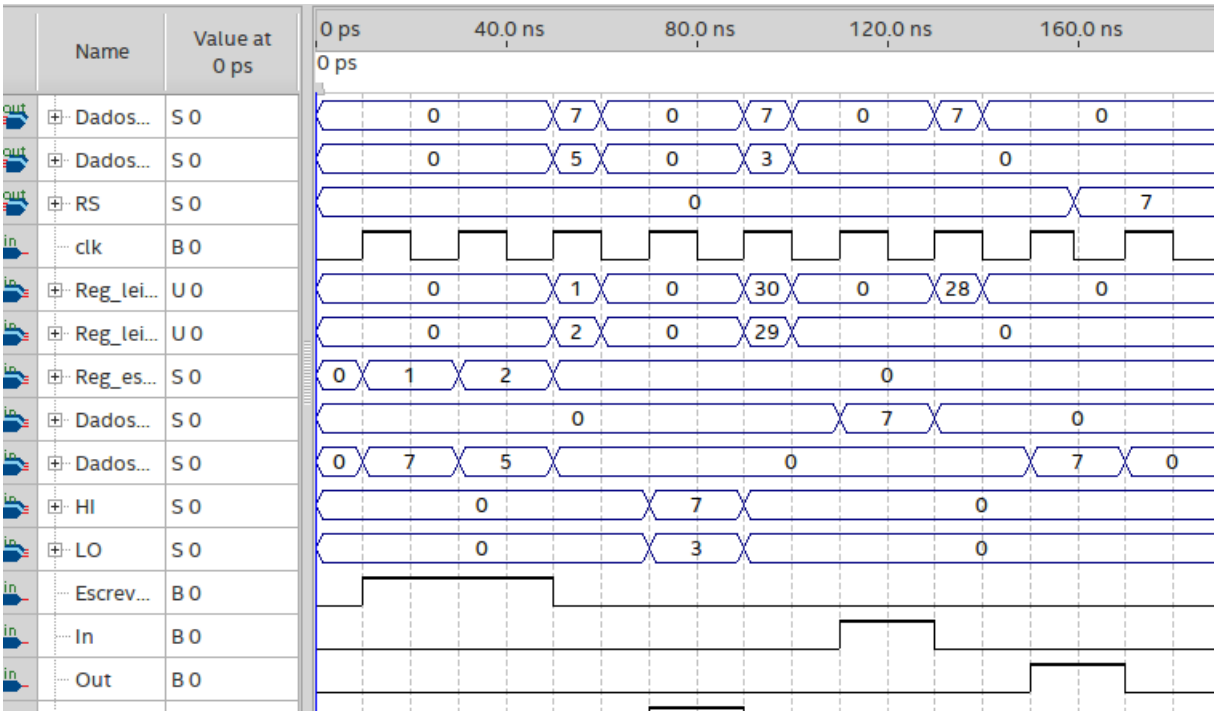
- Na próxima descida de *clock*, correponde ao momento em que o tempo é 80 ns, uma escrita nos registradores HI e LO é realizada. A entrada do sinal de controle "WriteHILO"acabou sendo cortada da imagem, mas ela fica alta entre os momentos correspondentes ao tempo em 70 ns a 90 ns. Os números escritos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

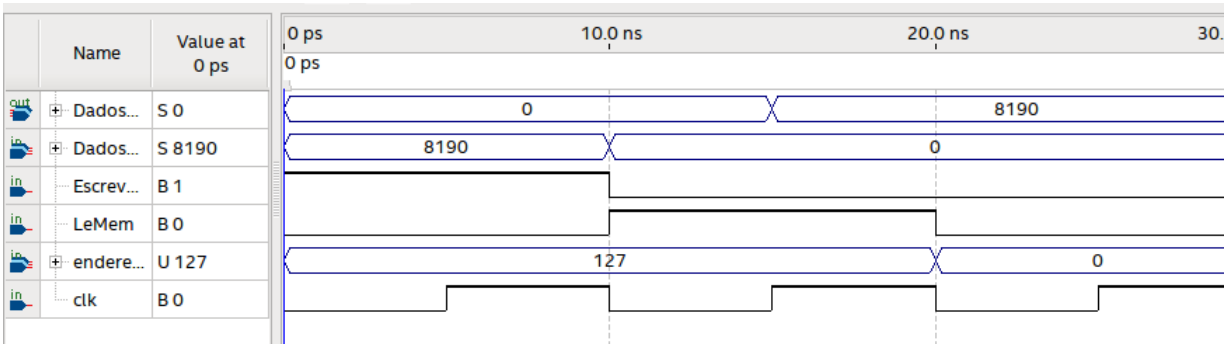
Figura 7 – Simulação em forma de onda realizada no banco de registradores.



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.

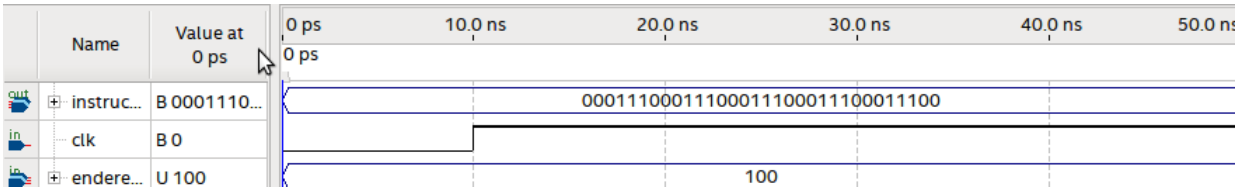
Figura 8 – Simulação em forma de onda da Memória de Dados.



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.

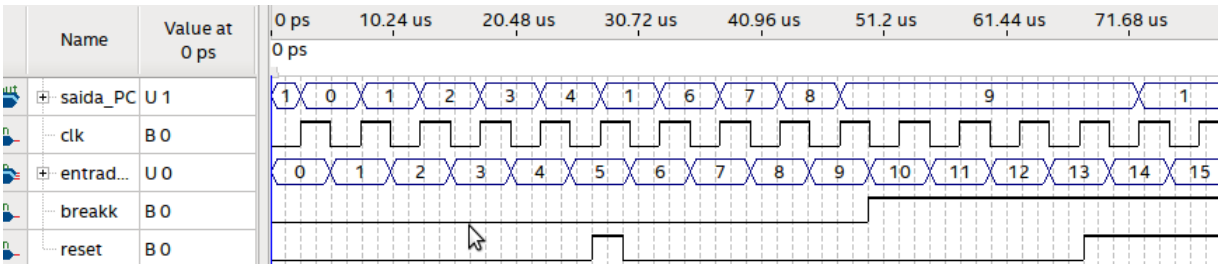
Figura 9 – Simulação em forma de onda realizada na Memória de Instruções.



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 o módulo de entrada . Sendo assim, uma vez que a simulação do "PC"confirmar seu funcionamento, não é necessário simular esse último, 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 10 – Simulação em forma de onda realizada no PC.

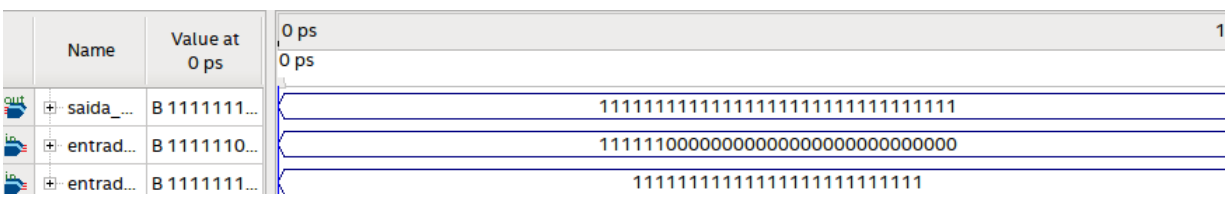


5.6 Simulação dos multiplexadores, do concatenador e do extensor

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".

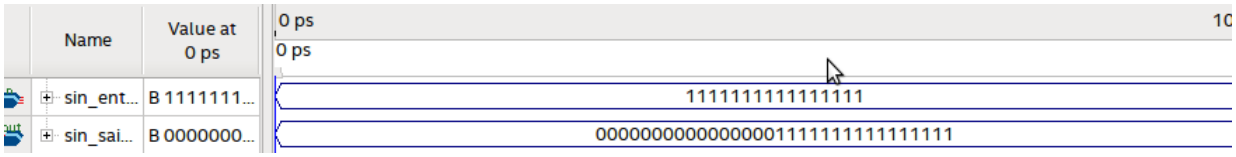
Para testar o concatenador as entradas foram preenchidas com números arbitrários e a simulação confirmou que, de fato, a concatenação foi realizada.

Figura 11 – 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.

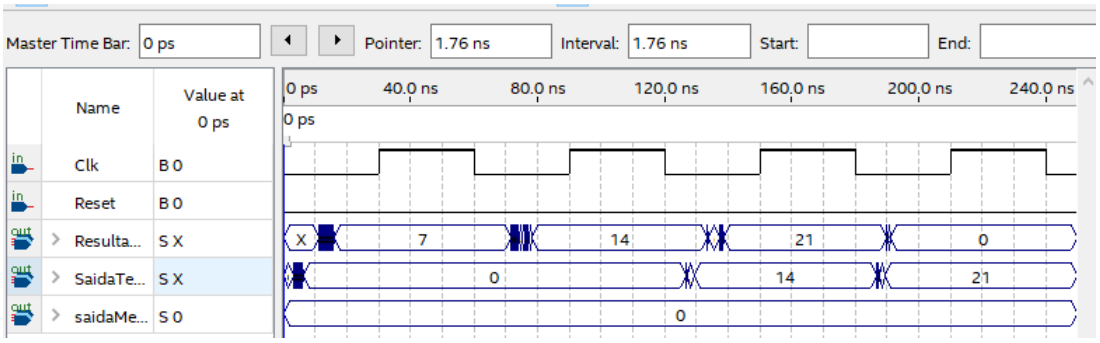
Figura 12 – Simulação em forma de onda do Extensor



5.7 Simulação do processador

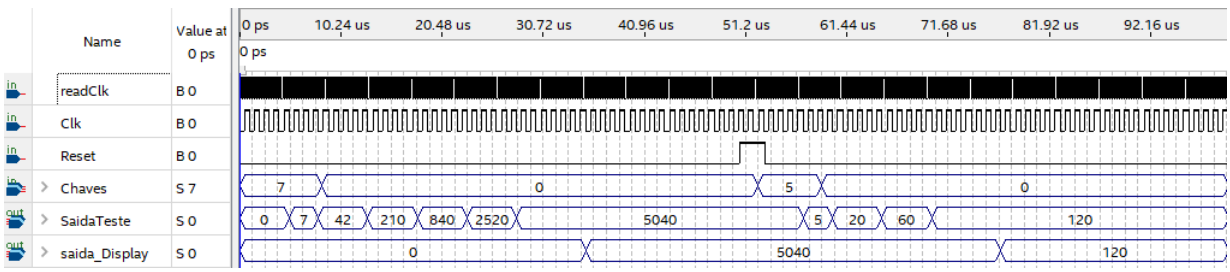
Para simular o processador funcionando, houveram dois testes principais, um mais simples, que consiste na realização de três somas de 7 consecutivas, e um mais complexo, onde houve a implementação do algoritmo de calcular o fatorial de um número, de acordo com o valor na entrada.

Figura 13 – Simulação de adições no processador.



Nesta primeira simulação, de cima pra baixo, tem-se o *clock*, o sinal de *reset*, responsável por fazer com que o o PC volte para o primeiro endereço de instrução. Depois, tem-se uma saída que denota os resultados dos calculos realizados pela ULA, e, por último, um valor escrito no banco de registradores, denotado pelo rótulo "SaídaTeste". Observe que a soma consecutiva de 7 três vezes apresenta o resultado esperado.

Figura 14 – Simulação de um algoritmo para calculo do fatorial de um número.



Nesta simulação, um sinal de *clock* extra foi adicionado na memória de dados, mais rápido que o original, logo abaixo. Denotado por "readClk"esse sinal é responsável por fazer com que a leitura na memória de dados ocorra mais rápido. Neste teste, tem-se as entradas denotadas pelo rótulo "Chaves"que representam os *switches* do FPGA, e a saída denotada por "saida_Display", que representa o valor que será apresentado nos *displays* de 7 segmentos do FPGA. Observe que foi realizado o cálculo do fatorial de 2 números. Primeiro, valor 7 é escrito na entrada "Chaves"e o processador calcula seu fatorial, de modo que o resultado seja coerente. Posteriormente, o valor de "Chaves"é mudado para 5, e o sinal *"reset"* é ativado, de modo com que o processador execute o fatorial novamente. Observe que o resultado obtido novamente foi coerente.

/sectionTestes realizados no FPGA

Esta seção possui por finalidade descrever como ocorreu o procedimento de simulação de dois algoritmos implementados através do uso linguagem de máquina desenvolvida durante o projeto. A implementação ocorreu de modo geral, através da escrita manual na memória de instruções.

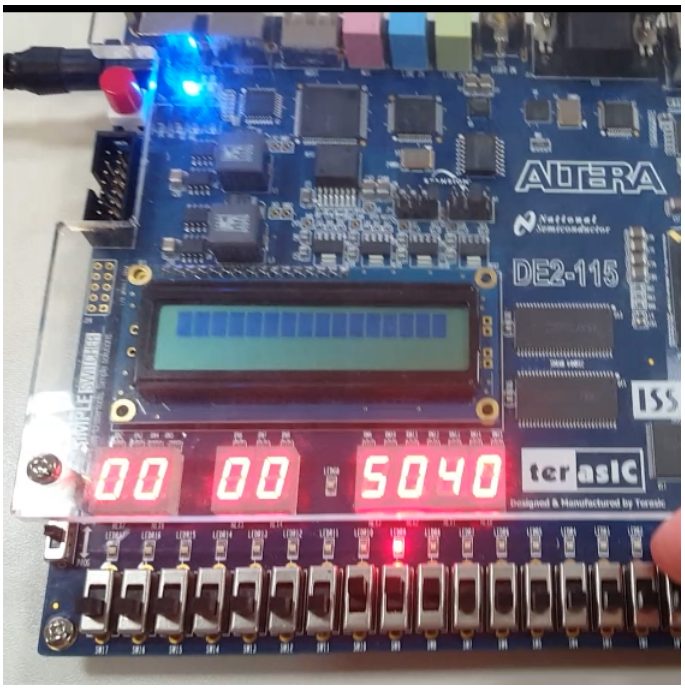
Simulado anteriormente, o primeiro algoritmo implementado foi o que realiza o cálculo do fatorial de um número inserido pelo usuário.

```
1  //FATORIAL
2                  mem_instrucao[2][31:26] = 6'b011110; //IN
3                  mem_instrucao[2][25:0] = 26'b0;
4
5                  mem_instrucao[3][31:26] = 6'b000010; //addi // UTILIZADO COMO LOAD
                       IMEDIATE
6                  mem_instrucao[3][25:21] = 5'b11010;     //registrador zero
7                  mem_instrucao[3][20:16] = 5'b00011;     //registrador 3 // destino
8                  mem_instrucao[3][15:0] = 16'd1;
9
10                 mem_instrucao[4][31:26] = 6'b000001; //add // utilizado como MOV
11                 mem_instrucao[4][25:21] = 5'b11010;     //registrador zero
12                 mem_instrucao[4][20:16] = 5'b11100;     //registrador RE
13                 mem_instrucao[4][15:11] = 5'b00001;     //registrador 1 // destino
14                 mem_instrucao[4][10:6] = 5'b0;
15
16                 mem_instrucao[5][31:26] = 6'b000100; //subi
17                 mem_instrucao[5][25:21] = 5'b11100;     //registrador RE
18                 mem_instrucao[5][20:16] = 5'b11100;     //registardor RE// destino
19                 mem_instrucao[5][15:0] = 16'd1;
20
21                 mem_instrucao[6][31:26] = 6'b000101; //mult
22                 mem_instrucao[6][25:21] = 5'b11100;
23                 mem_instrucao[6][20:16] = 5'b00001;
24                 mem_instrucao[6][15:0] = 16'b0;
25
26                 mem_instrucao[7][31:26] = 6'b000001; //add // utilizado como MOV
27                 mem_instrucao[7][25:21] = 5'b11010;     //registrador zero
28                 mem_instrucao[7][20:16] = 5'b11101;     //registrador LO
29                 mem_instrucao[7][15:11] = 5'b00001;     //registrador 1 // destino
30                 mem_instrucao[7][10:6] = 5'b0;
31
```

```
32                  mem_instrucao[8][31:26] = 6'b011001; //beq para o final do programa
33                  mem_instrucao[8][25:21] = 5'b11100;
34                  mem_instrucao[8][20:16] = 5'b00011;
35                  mem_instrucao[8][15:0] = 16'd1;
36
37                  mem_instrucao[9][31:26] = 6'b010111; // jump para la o do fatorial
38                  mem_instrucao[9][25:0] = 26'd5;
39
40                  mem_instrucao[10][31:26] = 6'b011101; //OUT
41                  mem_instrucao[10][25:21] = 5'b00001;
42                  mem_instrucao[10][20:16] = 5'b11010;
43                  mem_instrucao[10][15:0] = 16'b0;
44
45                  mem_instrucao[11][31:26] = 6'b011100; // BREAK
46                  mem_instrucao[11][25:0] = 26'd0;
```

Cada instrução teve seus campos escritos de forma individual, para evitar confusões quanto ao uso da linguagem de máquina, e facilitar a correção caso houvesse um erro de implementação.

Figura 15 – Cálculo do fatorial de 7 utilizando o processador desenvolvido e o dispositivo FPGA.



Vale ressaltar que as primeiras 10 chaves do dispositivo FPGA, da esquerda para a direita, servem para que seja realizada a entrada de um dado, de modo que a décima chave, contada da esquerda para a direita, manipula o bit menos significativo. As duas últimas chaves possuem a funcionalidade, consecutivamente, de *"Enter"*, para finalizar a inserção de um dado, e de *"Reset"*.

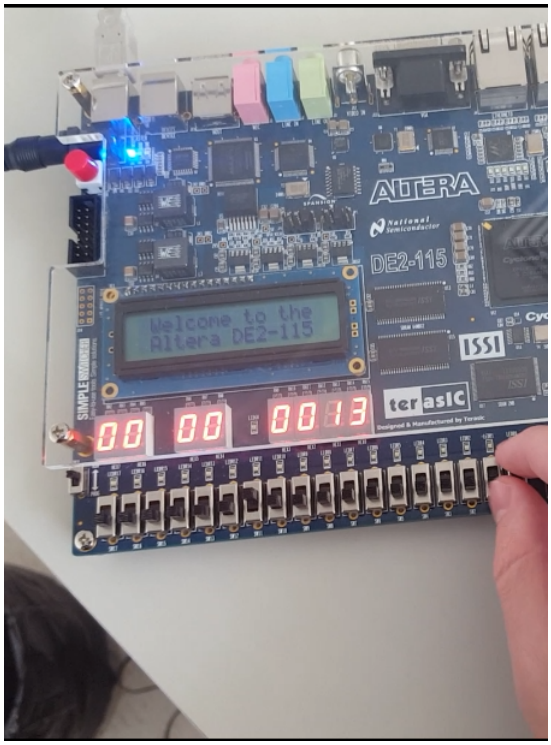
O segundo algoritmo implementado calcula o n-ésimo termo da sequência de fibonacci, onde n é informado pelo dado de entrada manipulado pelo usuário.

```
// FIBONACCI f(0) = 0  f(1) = 1 f(2) = 1 f(3) = 2

                mem_instrucao[2][31:26] = 6'b011110; //IN
                mem_instrucao[2][25:0] = 26'b0;

                mem_instrucao[3][31:26] = 6'b000010; //addi // UTILIZADO COMO LOAD
                    IMEDIATE
                mem_instrucao[3][25:21] = 5'b11010;     //registrador zero
                mem_instrucao[3][20:16] = 5'b00001;     //registrador 1 // destino
                mem_instrucao[3][15:0] = 16'd0;

                mem_instrucao[4][31:26] = 6'b000010; //addi // UTILIZADO COMO LOAD
                    IMEDIATE
                mem_instrucao[4][25:21] = 5'b11010;     //registrador zero
                mem_instrucao[4][20:16] = 5'b00010;     //registrador 2 // destino
                mem_instrucao[4][15:0] = 16'd1;

                mem_instrucao[5][31:26] = 6'b000100; //subi
                mem_instrucao[5][25:21] = 5'b11100;     //registrador RE
                mem_instrucao[5][20:16] = 5'b11100;     //registardor RE// destino
                mem_instrucao[5][15:0] = 16'd1;

                //INICIO DO LA 0//

                mem_instrucao[6][31:26] = 6'b000001; //add //
                mem_instrucao[6][25:21] = 5'b00001;     //registrador 1
                mem_instrucao[6][20:16] = 5'b00010;     //registrador 2
                mem_instrucao[6][15:11] = 5'b00011;     //registrador 3 // destino
                mem_instrucao[6][10:6] = 5'b0;

                mem_instrucao[7][31:26] = 6'b000100; //subi
                mem_instrucao[7][25:21] = 5'b11100;     //registrador RE
                mem_instrucao[7][20:16] = 5'b11100;     //registardor RE// destino
                mem_instrucao[7][15:0] = 16'd1;

                mem_instrucao[8][31:26] = 6'b000001; //add // utilizado como MOV
                mem_instrucao[8][25:21] = 5'b11010;     //registrador ZERO
                mem_instrucao[8][20:16] = 5'b00010;     //registrador 2
                mem_instrucao[8][15:11] = 5'b00001;     //registrador 1 // destino
                mem_instrucao[8][10:6] = 5'b0;

                mem_instrucao[9][31:26] = 6'b000001; //add // utilizado como MOV
                mem_instrucao[9][25:21] = 5'b11010;     //registrador ZERO
                mem_instrucao[9][20:16] = 5'b00011;     //registrador 3
                mem_instrucao[9][15:11] = 5'b00010;     //registrador 2 // destino
                mem_instrucao[9][10:6] = 5'b0;

                mem_instrucao[10][31:26] = 6'b011001; //beq para o final do programa
                mem_instrucao[10][25:21] = 5'b11010;
                mem_instrucao[10][20:16] = 5'b11100;
                mem_instrucao[10][15:0] = 16'd1;

                mem_instrucao[11][31:26] = 6'b010111; // jump para la o do fatorial
                mem_instrucao[11][25:0] = 26'd6;

                mem_instrucao[12][31:26] = 6'b011101; //OUT
                mem_instrucao[12][25:21] = 5'b00011;    // Registrador 3
                mem_instrucao[12][20:16] = 5'b11010;
                mem_instrucao[12][15:0] = 16'b0;

```

```
59                  mem_instrucao[13][31:26] = 6'b011100; // BREAK
60                  mem_instrucao[13][25:0] = 26'd0;
```

Esse código possui uma quantidade maior de instruções do que o primeiro.

Figura 16 – Cálculo do sétimo termo da sequência de Fibonacci utilizando o processador desenvolvido e o dispositivo FPGA.



6 Considerações Finais

O ponto crucial da primeira etapa do desenvolvimento do projeto foi a tomada de decisão sobre os aspectos e características da arquitetura base. Antes da ideia de se basear na arquitetura MIPS, houve um breve estudo sobre a arquitetura ARM, especificamente da implementação que recebeu o nome de Amber, baseada no ARMv2, umas das versões mais antigas e simples desta arquitetura.

Em um segundo momento, percebeu-se que a ideia de implementar um processador baseado na arquitetura ARM ia se tornar muito complicada e trabalhosa, e assim, optou-se pela arquitetura MIPS, que já foi estudada em disciplinas anteriores.

Depois dessa primeira decisão foram definidos os aspectos da arquitetura desenvolvida, utilizando as características do MIPS, e , depois disso, todos os aspectos técnicos foram definidos:

- O conjunto de instruções foi definido.
- Os modos de endereçamento foram escolhidos.
- Os formatos de instruções foram definidos.
- Os registradores e suas funções contidos no banco de registradores foram definidos.
- Através do uso de todas essas características, um primeiro caminho de dados foi desenvolvido.

Pode-se afirmar que o produto desta primeira etapa do projeto foi o caminho de dados, utilizado como "planta"do processador, durante a implementação. Pelo fato da primeira etapa neste projeto se tratar de tomadas de decisão e revisão de conceitos teóricos, não houveram maiores problemas ou dificuldades.

A principal dificuldade encontrada na segunda 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 desta etapa de modo satisfatório.

Depois da primeira etapa, foi estudado um modo de interação do processador com dispositivos externos da placa utilizada. E assim, os procedimentos de entrada e 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.

E, finalmente, houve a implementação da unidade de controle, e a posterior interligação dos blocos desenvolvidos. As simulações realizadas foram fundamentais na resolução dos erros e equívocos que aconteceram durante a implementação. Entre eles, houveram confusões com a atribuição do valor de cada sinal de controle para cada instrução, e com os nomes das variáveis utilizadas, mas que foram apropriadamente resolvidas.

Vale ressaltar que o processador foi implementado de modo que seja possível realizar uma posterior modificação ou inclusão de novas instruções ou de novos módulos, assim como é possível implementar um pipeline.

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 16.