UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

IMPLEMENTAÇÃO DA M++ EM FPGA

ANDRÉ LEONARDO BIEGING

BLUMENAU
2018

ANDRÉ LEONARDO BIEGING

IMPLEMENTAÇÃO DA M++ EM FPGA

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Miguel A. Wisintainer, Orientador

BLUMENAU
2018

IMPLEMENTAÇÃO M++ EM FPGA

Por

ANDRÉ LEONARDO BIEGING

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof(a). Miguel Alexandre Wisintainer – Orientador, FURB
Membro:	Prof(a). Mauro Marcelo Mattos – FURB
Membro:	Prof(a). Francisco Adell Péricas – FURB

Blumenau, 11 de dezembro de 2018

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me dado tudo o que tenho hoje e por me proporcionar a inteligência para desenvolver este trabalho. Agradeço à minha noiva por sempre me ajudar e me incentivar a dar o meu melhor. Também agradeço aos meus pais por me proporcionarem a oportunidade de fazer uma faculdade. Por último, mas não menos importante, agradeço ao meu orientador, Miguel A. Wisintainer, por toda a ajuda e incentivo durante o desenvolvimento.

RESUMO

O presente trabalho tem como objetivo implementar a arquitetura da M++ na plataforma FPGA. Foi utilizada a linguagem de descrição de hardware Verilog. Para desenvolver o trabalho, primeiro foram levantados os requisitos funcionais e não funcionais necessários, para então criar a especificação do trabalho, que é uma descrição detalhada da arquitetura da M++. Para a implementação da especificação, foi utilizada a IDE Quartus para desenvolver e compilar o código Verilog e a sua extensão TimeQuest Timing Analyzer para fazer o cálculo de tempos, enquanto o software ModelSim*-Intel® FPGA edition foi utilizado para a simulação do design. O programa MontadorMmaismais foi necessário para desenvolver os programas para os testes, e o programa MapReader foi desenvolvido para converter arquivos `.map` em programas em Verilog. O resultado do desenvolvimento foi um design capaz de executar as instruções definidas pela especificação, e tem uma frequência máxima de execução muito maior do que o que é possível em relação ao software de simulação original da M++. Com base nos resultados do trabalho, conclui-se que é possível implementar a arquitetura da M++ em Verilog e conseguir um design com uma performance considerável.

Palavras-chave: Arquitetura. Verilog. FPGA. Simulação.

ABSTRACT

This project's objective is to implement de M++ architecture in the FPGA platform. The hardware description language Verilog was used. The first step to develop the project was to list the needed functional and non-functional requisites, to then create the project's specification, which is a detailed description of the M++ architecture. For the specification's implementation, the Quartus IDE was used to develop and compile the code and its TimeQuest Timing Analyzer extension calculated the design timings, while the software ModelSim*-Intel® FPGA edition was used to simulate the design. The program MontadorMmaismais was necessary to develop the test programs and the program MapReader was developed to convert .map files into Verilog programs. The final result was a design capable of executing the instructions defined by the specification and has a maximum working frequency that's much higher than what is possible with the M++'s current simulation software. Based on the achieved results, it's concluded that it's possible to implement the M++'s architecture in Verilog and achieve a design with a considerable performance.

Key-words: Architecture. Verilog. FPGA. Simulation.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ASIC – Application-Specific Integrated Circuits

ASSP – Application-Specific Standard Products

FPGA – Field Programmable Gate Array

RAM – Random Access Memory

RISC – Reduced Instruction Set Computer

ROM – Read Only Memory

ULA – Unidade de Lógica e Aritmética

SUMÁRIO

1 INTRODUÇÃO

As tecnologias dos microprocessadores vêm avançando rapidamente desde a invenção do transistor em 1947. De *mainframes* gigantes capazes de realizar até 3 milhões de cálculos por segundo (VAUGHAN-NICHOLS, 2018), a minúsculos *chips* contendo bilhões de transistores e *racks* de servidores capazes de executar quadrilhões de operações por segundo e transferir dados a taxas gigantescas (CUTRESS, 2018). Atualmente, estes *chips* estão divididos em três categorias: Application-Specific Standard Products (ASSPs), Application-Specific Integrated Circuits (ASICs) e os Field Programmable Gate Array (FPGAs).

Os ASICs são componentes criados para uma única aplicação e são desenvolvidos para serem usados, normalmente, por apenas uma empresa. Os ASSPs são desenvolvidos para aplicações de uso geral, como os processadores dos nossos computadores, e são vendidos para uma quantidade maior de clientes. Mas os ASICs e ASSPs tem suas limitações. Para criá-los é necessário uma equipe de engenheiros capazes e de grandes recursos financeiros. Além de demandar um tempo de desenvolvimento chegando a 1 ano (MURALI, 2013), pequenos erros ou mudanças de arquitetura podem levar o projeto à estaca zero, pois, uma vez que lançado, o produto não pode mais sofrer modificações de hardware.

Os FPGAs, segundo Moore (2017, p. 4, tradução nossa), “dispositivos semicondutores cuja função pode ser definida após sua fabricação, [...] podendo ser modificada [...] para atender a aplicações específicas mesmo após o produto estar instalado no campo”. Esta definição mostra que um FPGA é muito mais flexível do que os ASSPs e ASICs, já que erros podem ser corrigidos e funções podem ser adicionadas facilmente. Além disso, esta tecnologia é muito mais barata por possuir um tempo de desenvolvimento menor e por haver kits de desenvolvimento bem acessíveis.

Com base no que foi apresentado, este trabalho propõe a implementação de um microprocessador na plataforma FPGA. A arquitetura será baseada na M++ (BORGES, 2003), um processador criado em um software de simulação de lógica. Para isso será preciso traduzir corretamente a arquitetura de uma plataforma digital para uma física, resolver problemas específicos relacionados ao desenvolvimento no hardware e aprender o uso correto desta tecnologia, por ser bastante complexa.

1.1 OBJETIVOS

O objetivo deste trabalho é portar a M++ para um FPGA.

Os objetivos específicos são:

a) carregar programas pré-gravados na memória do microprocessador;

b) ler entradas e produzir saídas conforme cada programa é executado;
c) verificar a possibilidade de adicionar instruções extras da M+++ à arquitetura e verificar viabilidade de novas instruções;
d) criar documentação usada como material de referência do assunto na disciplina.

1.2 ESTRUTURA

O trabalho começa com uma seção de apresentação ao tema através de trabalhos correlatos e uma introdução à linguagem Verilog. Em seguida a M++ é apresentada como o trabalho base, e suas funcionalidades são brevemente descritas. Na seção de desenvolvimento, são apresentados os requisitos do trabalho, uma especificação detalhada da arquitetura da M++, como a especificação foi implementada em Verilog e quais foram os principais programas e ferramentas utilizadas durante o projeto. Para finalizar, é mostrada como funciona a operação de um programa através do simulador e quais foram os resultados alcançados com o trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

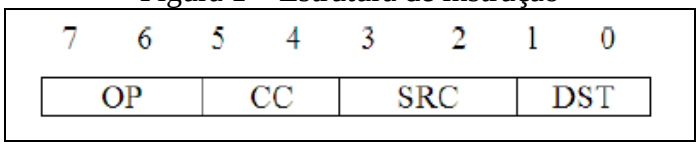
Esta seção visa criar um embasamento teórico sobre o assunto. Para isso, foram encontrados trabalhos similares ao que será proposto. Os trabalhos de Ayeh et al. (2008) e Pablo et al. (2016), descrevem uma pesquisa sobre o desenvolvimento de uma CPU em um FPGA. O terceiro outro CPU de 8 bits implementada em FPGA voltada para o ensino (ZALAVA et al., 2015). Também serão discutidos alguns detalhes da linguagem Verilog.

2.1 FPGA IMPLEMENTATION OF AN 8-BIT SIMPLE PROCESSOR

É um processador simples de 8 bits, desenvolvido por Ayeh et al., (2008). Consiste em 4 registradores de 4 bits, uma memória capaz de armazenar 16 palavras de 8 *bits*, uma unidade de controle e uma unidade de lógica e aritmética. A CPU foi programada e testada em um FPGA Xilinx Spartan 3 utilizando a linguagem VHDL e a IDE ISE Foundation 8.1.

Possui um set de instruções bem simples, sendo que os 4 *bits* mais altos são *bits* de controle e os 4 *bits* mais baixos são bits de endereço. A Figura 1 mostra a distribuição dos bits em uma instrução. OP são *bits* para determinar a operação. CC são *bits* para determinar o valor do bit de *carry*. SRC são bits de endereço da origem dos dados. DST são *bits* de endereço do destino dos dados processados.

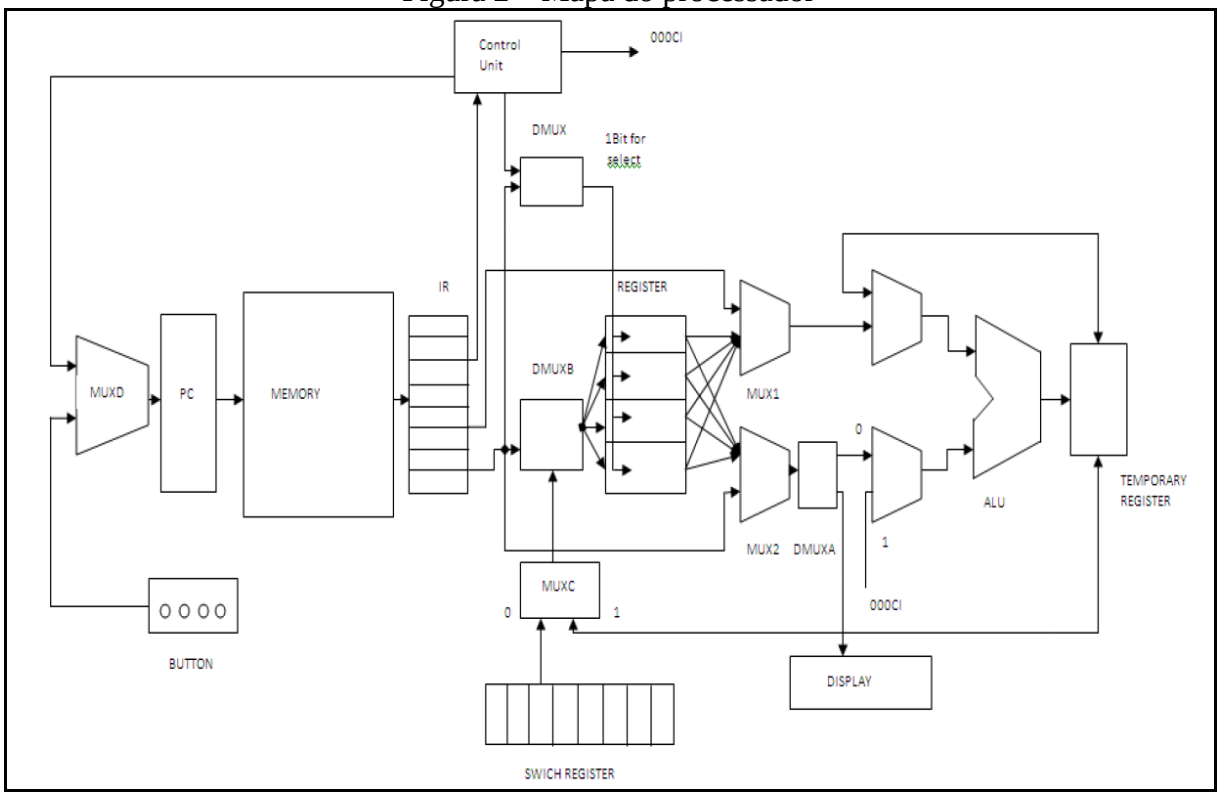
Figura 1 – Estrutura de instrução



Fonte: Ayeh et al., (2008).

As operações possíveis são soma e subtração com os valores armazenados nos endereços providos pelos *bits* SRC e DST. A Figura 2 mostra o mapa completo de todos os componentes internos da CPU: a unidade de controle, a unidade de lógica e aritmética, os registradores para realizar as operações, a memória e um botão para que o usuário controle a CPU.

Figura 2 – Mapa do processador



Fonte: AYEH et al., (2008).

Todas as partes individuais do processador foram testadas antes que fossem combinadas e testadas como um todo. Foi atingida uma frequência máxima de operação de 95.364MHz e menos de 10% dos elementos lógicos do FPGA foram utilizados.

2.2 A VERY SIMPLE 8-BIT RISC PROCESSOR FOR FPGA

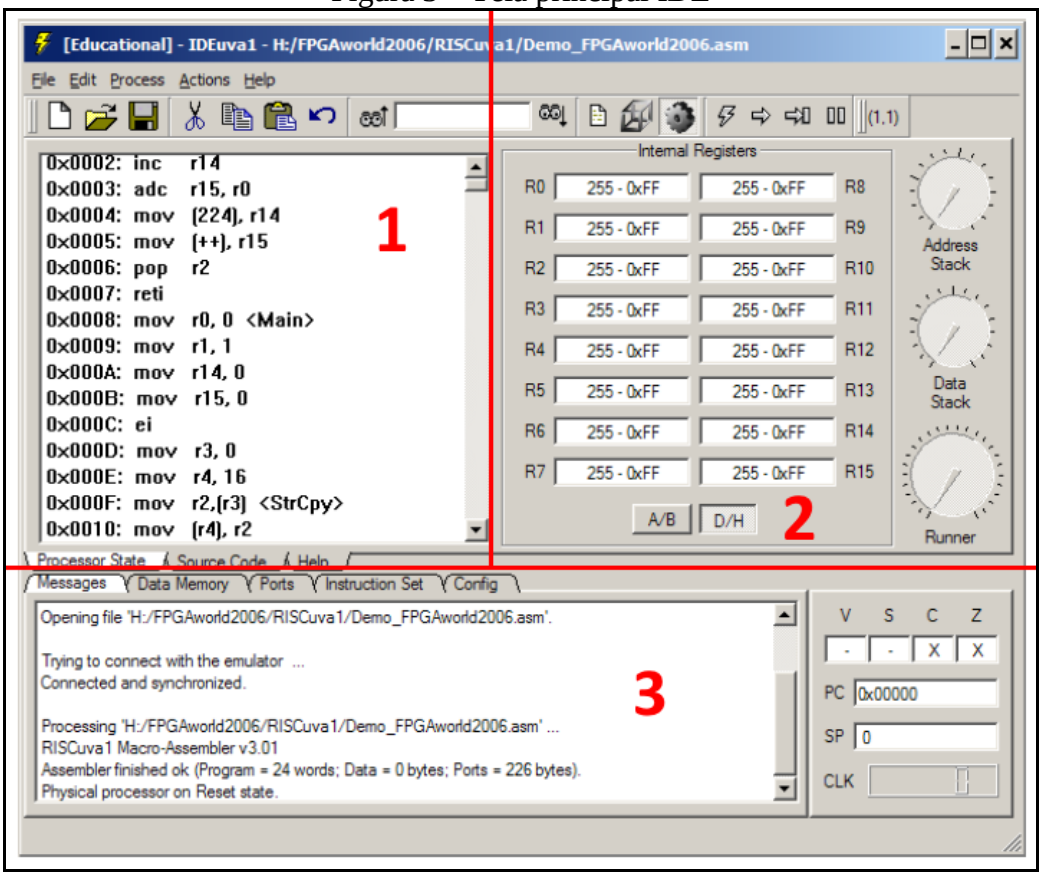
O RISCuva1, é um processador *Reduced Instruction Set Computer* (RISC) de 8 *bits* que utiliza a arquitetura Harvard, desenvolvido por PABLO et al., (2016), programado e testado em um FPGA Xilinx Spartan 6 utilizando a linguagem Verilog.

Contém 16 registradores internos de 8 *bits* e possui um set de instruções de 14 *bits* que vêm da memória do programa. Pode executar 12 operações nativas (aritmética, lógica e rotação de *bits*), suporta macros de operações, expressões *goto*, chamadas de sub-rotinas e chamadas de retorno para as mesmas. Também há uma pilha interna que possibilita uma chamada de até 16 sub-rotinas recursivamente. Fazem parte das características do *set* de instruções executar instruções em apenas um ciclo de *clock* e a Unidade de Lógica e Aritmética (ULA) pegar dois valores dos registradores e retornar o resultado para outro registrador, entre outros.

Uma IDE foi desenvolvida para este trabalho para criar programas, compilar e gravar os programas. A Figura 3 mostra uma tela da IDE. A área 1 da imagem mostra o código,

escrito em *assembly*, que será programado na CPU. A área 2 mostra os valores armazenados nos registradores internos. A área 3 é a área de mensagens da IDE, informando ao usuário dados sobre a compilação e programação.

Figura 3 – Tela principal IDE



Fonte: PABLO et al., (2016).

2.3 DESIGN OF A GENERAL PURPOSE 8-BIT RISC PROCESSOR FOR COMPUTER ARCHITECTURE LEARNING

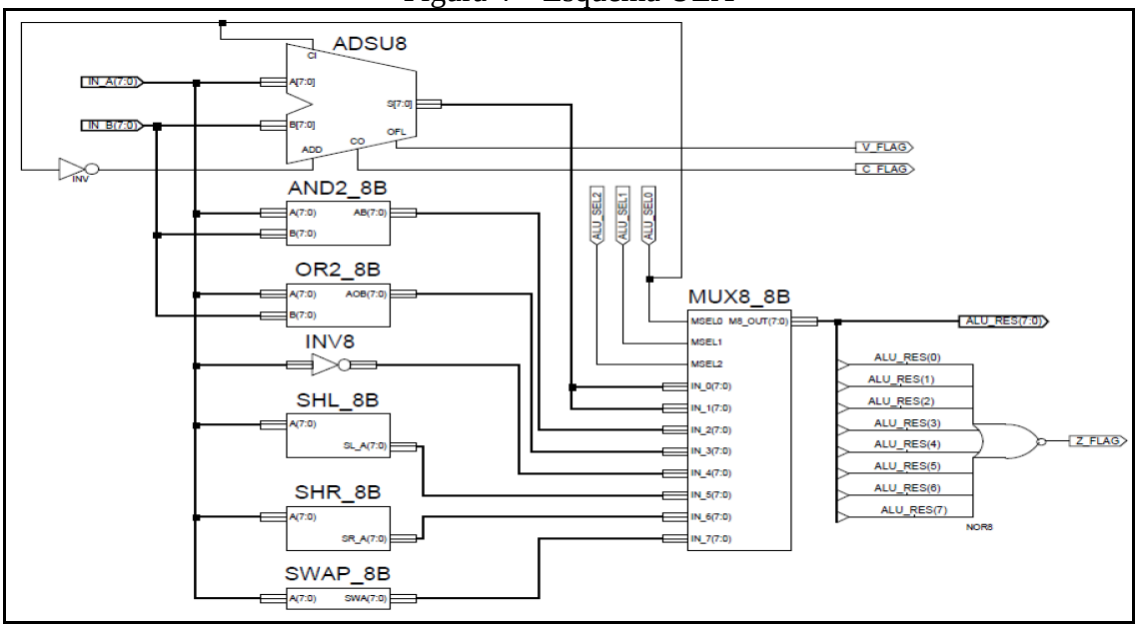
Esta CPU, desenvolvida por Zalava et al., (2015), é um processador RISC de 8 *bits*, foi implementado e testado em um FPGA Xilinx Spartan 3, utilizando as ferramentas XilinxISE na versão 14.2. Utiliza a arquitetura Harvard e tem um set de instruções RISC com 29 instruções. Possui instruções de soma e subtração, operações lógicas, chamadas de sub-rotinas e de retorno de sub-rotinas e há operações de *pop* e *push* para uma pilha de dados. Contém 8 registradores de 8 *bits*, 256 posições de memória *Read Only Memory* (ROM) de 16 *bits* para armazenar o programa e possui 256 posições de memória *Random Access Memory* (RAM) para armazenar dados. Também há uma ULA que executa duas operações aritméticas simples e seis operações lógicas.

Foi testado um programa, com sucesso, que calcula os primeiros 10 elementos da sequência de Fibonacci e os armazena em posições consecutivas de memória. Também foi

observado que o processador ocupa um espaço de menos de 5% dos elementos lógicos disponíveis.

A Figura 4 mostra como é formada a unidade de lógica e aritmética desta CPU. Possui a entrada de dados de 2 registradores, que são ligados à entrada de dados de cada um dos blocos de operações da ULA. O resultado de cada operação é colocado nas entradas de um multiplexador. A CPU então escolhe qual dos resultados será lido.

Figura 4 – Esquema ULA



Fonte: Zalava et al., (2015).

2.4 VERILOG

Esta seção introduz alguns tópicos básicos e importantes sobre a linguagem Verilog para que os códigos apresentados sejam melhor compreendidos.

2.4.1 Módulos

> Conforme Palnitkar (tradução nossa, 2003, p. 30), um módulo é o bloco base de construção em Verilog. Um módulo pode ser um elemento ou uma coleção de blocos de design de mais baixo nível. Tipicamente, elementos são agrupados em módulos para oferecer funções comuns que são utilizadas em vários lugares do design. Um módulo fornece a funcionalidade necessário a um bloco de nível mais alto através da sua interface de entradas e saídas, mas esconde seu funcionamento interno.

O Quadro 1 mostra a sintaxe para definir um módulo. Após a palavra reservada `module`, é definido o nome do módulo. Em seguida, entre parênteses e finalizado por um ponto e vírgula, é definida a lista de terminais, que é uma lista de nomes separados por virgulas. A lista de terminais define a interface do módulo com um bloco de nível mais alto.

Toda a funcionalidade do módulo é escrita após a lista de terminais e antes da palavra reservada `endmodule`, que finaliza a declaração do módulo.

Quadro 1 – Sintaxe de definição de um módulo

```
module <module_name> (<module_terminal_list>);

...
<module internals>
...
...
endmodule
```

Fonte: Palnitkar (2003).

2.4.2 Bloco de estímulo / Testbench

Para fazer testes em um design é necessário definir um bloco de estímulo. Neste bloco é criada a instância do módulo de nível mais alto, e são gerados valores para suas entradas. Com um bloco de estímulo, é possível fazer uma simulação completa do modelo desenvolvido com o software ModelSim*-Intel® FPGA edition. Durante a simulação, é possível fazer o *debug* do código e ver as formas de ondas de todos os sinais em uma linha do tempo.

2.4.3 Entradas e Saídas

Em um módulo, é necessário definir se os parâmetros da lista de terminais são entradas ou saídas. Para definir um parâmetro como entrada é utilizada a palavra reservada `input` seguida pelo nome do parâmetro, e para definir como saída, é utilizada a palavra reservada `output` seguida pelo nome do parâmetro. O Quadro 2 mostra a definição de entradas e saídas de um módulo.

Quadro 2 – Definição das entradas e saídas

```
module ram_decode
    (
        en,
        addr,
        data
    );

    input en;
    input [3:0] addr;

    output [7:0] data;
```

Fonte: elaborado pelo autor.

2.4.4 Tipos de Variáveis

Há dois tipos principais, as variáveis do tipo `reg`, e as do tipo `wire`. Variáveis do tipo `reg` são registradores e podem armazenar valores pelas expressões de atribuição. As variáveis do tipo `wire` são fios, e podem apenas ligar um ponto ao outro. Para conectar um `wire` à uma parte do circuito é utilizado um bloco `assign`.

O Quadro 3 mostra o uso do tipo `wire` para criar o módulo `and_gate`. A variável `and_temp` recebe o resultado no bloco `assign` e em seguida é atribuída à saída `out1`. Neste caso a variável não armazena o resultado, ela apenas conecta o resultado da operação à saída.

Quadro 3 - Tipo `wire` em um módulo

```
module and_gate
    (
        in1,
        in2,
        out1
    );

    input in1, in2;

    output out1;

    wire and_temp;

    assign and_temp = in1 & in2;

    assign out1 = and_temp;

endmodule
```

Fonte: elaborado pelo autor.

O Quadro 4 mostra o uso do tipo `reg` para criar o mesmo módulo `and_gate`. Neste caso, a variável `out1` sempre armazena o resultado da operação.

Quadro 4 – Tipo reg em um módulo

```
module and_gate
    (
        in1,
        in2,
        out1
    );

    input in1, in2;

    output out1;

    reg out1;

    always @ (in1 or in2)
        begin
            out1 = in1 & in2;
        end

endmodule
```

Fonte: elaborado pelo autor.

2.4.5 Delay

Para evitar a *Race Condition*, é adicionado um *delay* com o símbolo # seguido pelo valor do atraso. É importante notar que esta função é utilizada apenas para a simulação. O Quadro 5 mostra o uso do atraso para gerar a oscilação do sinal `clk` em um bloco `always`. Como não há lista de sensibilidades no bloco `always`, este será executado repetidamente para sempre. Toda vez que é executado, há um atraso de 10 unidades de tempo antes de o valor do `clk` ser invertido, fazendo com que o valor do `clk` oscile com um período de 20 unidades de tempo.

Quadro 5 - Uso do atraso para gerar sinal do clk

```
always
    begin
        #10 clk = !clk;
    end
```

Fonte: elaborado pelo autor.

2.4.6 Blocos `initial`

O valor padrão das variáveis em Verilog é ter o valor indeterminado e por isso é ter uma forma de inicializar seus valores. O bloco `initial` é principalmente utilizado para esta função, e é executado apenas uma vez na inicialização da simulação. O Quadro 6 mostra o bloco `initial` atribuindo valores padrão para a variável `data`.

Quadro 6 - Bloco initial inicializando variáveis

```
initial
    begin
        data = 8'b00000000;
    end
```

Fonte: elaborado pelo autor.

2.4.7 Blocos *always*

Funcionam como um evento. Possuem uma lista de sensibilidades, e sempre que algum valor da lista é alterado, o bloco é executado. A sintaxe para o bloco `always` é: `always @(<lista_sensibilidades>)`, onde `lista_sensibilidades` pode ter diversos valores, conforme o Quadro 7. Na primeira variação, não há lista de sensibilidades, o que significa que o bloco será executado repetidamente em um *loop* infinito. Ao colocar o nome de uma variável, o bloco será executado sempre que seu valor mudar. Pode ser adicionado o prefixo `posedge` ou `negedge` para que o bloco execute somente na borda de subida e de descida, respectivamente, da variável. Também é possível adicionar várias variáveis à lista e criar uma lógica de acionamento mais complexa.

Quadro 7 – Modelos de bloco `always`

```
always
    begin
        clk = ~clk;
    end

always @ (var1)
    begin
        out = var1;
    end

always @ (posedge var1)
    begin
        out = var1;
    end

always @ (var1 or var2)
    begin
        out = var1 & var2;
    end
```

Fonte: elaborado pelo autor

2.4.8 Expressão `assign`

É utilizada para conectar variáveis do tipo `wire` a outros sinais. O símbolo de atribuição `=` funciona apenas para variáveis do tipo `reg`, já que variáveis do tipo `wire` não podem armazenar valores.

2.4.9 Sintaxe Variável

Nas linguagens mais comuns, C++, ao declarar uma variável do tipo inteiro, por exemplo, são alocados 32 *bits* de memória para armazenar este dado. Em Verilog, é possível especificar qual o tamanho em *bits* de cada variável, utilizando o número de *bits* entre colchetes, entre o tipo da variável e seu nome, conforme o Quadro 8. Especificar o tamanho em *bits* da variável é opcional, e quando omitido, declara a variável com apenas 1 *bit*. É importante notar que em Verilog é possível definir a *endianess* de cada variável ao alterar a ordem dos números dentro dos colchetes. A variável declarada no quadro é do tipo *little endian*. É possível criar uma variável do tipo *big endian* ao declarar a variável como `reg [0:7] var2`.

Quadro 8 – Sintaxe de declaração de uma variável

```
<tipo> [<tamanho>] <nome>;

reg [7:0] var1;
```

Fonte: elaborado pelo autor.

2.4.10 Sintaxe *Array*

Conforme o Quadro 9 abaixo, para criar um *array*, a sintaxe é escrever o tipo, seguido pelo nome da variável e seguido pelo tamanho do *array* entre colchetes. Assim como a

declaração das variáveis, é possível omitir a parte `[<tamanho_vetor>]`. A variável criada no exemplo do quadro é um *array* que armazena 16 registradores de 8 *bits*. É comum criar o *array* como *big endian*, e as variáveis como *little endian*, assim como no exemplo.

Quadro 9 – Sintaxe de declaração de array

```
<tipo> [<tamanho_vetor>] <nome> [<tamanho_array>];
reg [7:0] var1 [0:15];
```

Fonte: elaborado pelo autor.

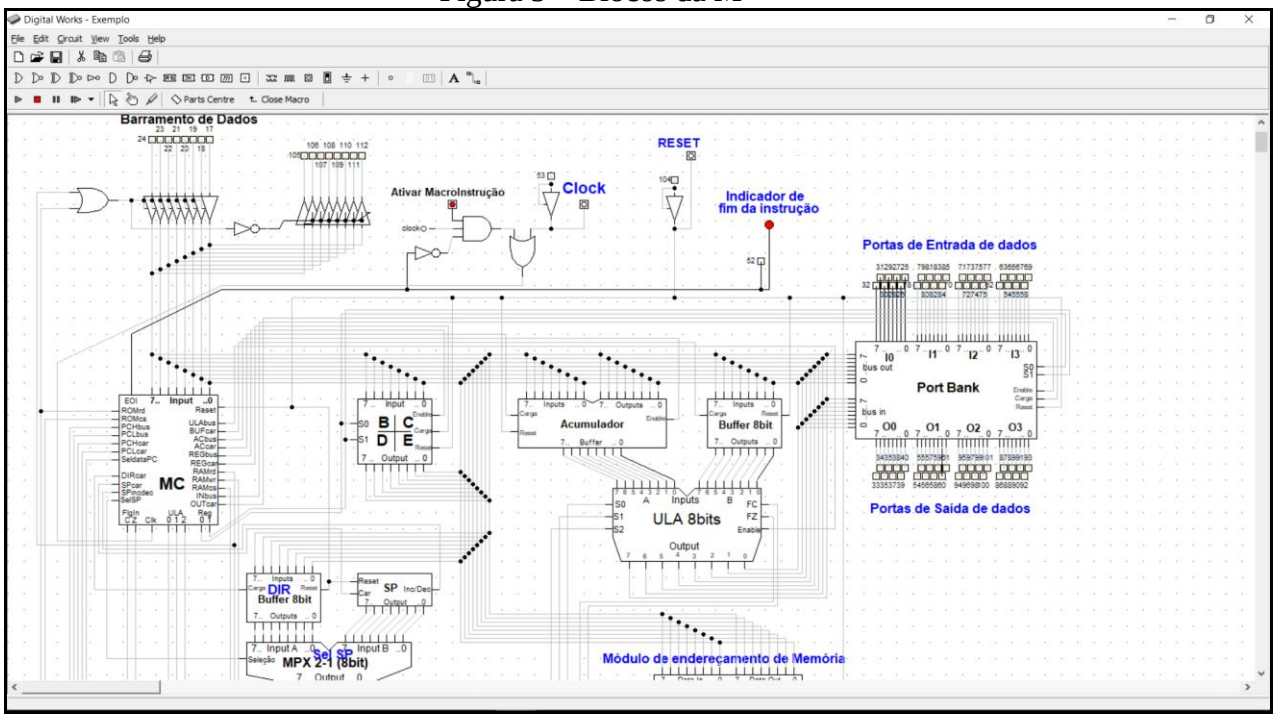
3 ARQUITETURA ATUAL

A Máquina++, desenvolvida por Borges, (2003), tem como objetivo mostrar o funcionamento de uma CPU hipotética através de um software de simulação. É capaz de executar cálculos, operações lógicas, movimentação de dados, chamadas de sub-rotinas e operações de entradas e saídas. Sua arquitetura contém diversas partes principais, sendo elas:

a) unidade de lógica e aritmética: é responsável por ler dados do acumulador e de algum registrador, e realizar operações lógicas e aritméticas com estes. As operações possíveis são: soma, subtração, `AND`, `OR`, `XOR` e `NOT`;
b) acumulador: responsável por conter um dado de 8 *bits* temporariamente até que este seja utilizado por algum outro bloco;
c) registradores: contém 4 células de memória capazes de armazenar um valor de 8 *bits*. Estes valores podem ser utilizados para realizar operações na ULA ou podem ser mostrados nas saídas;
d) barramento de dados: responsável por transferir os dados entre todos os blocos do microprocessador. Pode transferir até 8 *bits* simultaneamente;
e) banco de Portas: utilizado para realizar operações de E/S. Pode ler entradas e acionar saídas;
f) memória ROM: é uma memória de apenas leitura onde é armazenado o programa, que é lido pelo bloco módulo de controle;
g) módulo de controle: é o coração do projeto e é a parte que comanda todos os outros blocos. Ele lê a memória ROM e processa seus dados, passando por um decodificador de instruções que aciona as memórias de controle, que possuem as micro instruções do processador. Conforme são acessadas as memórias de controle, sinais são acionados para controlar os diversos blocos da Máquina ++.

A figura 5 mostra uma tela da M++, na qual aparecem a maioria dos blocos descritos acima: barramento de dados, microcontrolador, registradores, acumulador, ULA e banco de portas. Também há a entrada de *clock* e um botão de reset, utilizado para reiniciar a CPU manualmente.

Figura 5 – Blocos da M++



Fonte: Borges (2003).

Para seu desenvolvimento, o autor da M++ utilizou o software de simulação de lógica Digital Works. Este programa possibilita ao usuário criar diversos circuitos lógicos, combinando vários componentes e blocos que já vêm prontos. Também é possível escolher a velocidade da simulação e fazer o *debug* do sistema executando *clock* por *clock*, para melhorar a capacidade de análise. O projeto final é capaz de ler uma memória externa, que possui um programa, decodificar as instruções e acionar os sinais que comandam os outros blocos através das memórias de controle.

Com base na M++, foi desenvolvida outra máquina hipotética com algumas melhorias em relação a M++, e esta foi chamada de M+++. Esta foi criada por um aluno da turma de arquitetura de computadores.

As principais diferenças do novo modelo são: armazenamento de dados e armazenamento da pilha em memórias separadas, instruções de *push* e *pop* e também a possibilidade de salvar um dado na memória sem fornecer um endereço absoluto da memória, podendo ser salvo referenciando um registrador ao invés do endereço absoluto da memória.

4 DESENVOLVIMENTO DO PROCESSADOR

Para o desenvolvimento do processador, primeiro foram levantados os requisitos necessários para definir quais as funções desejadas para o projeto. Em seguida foi feita a especificação da arquitetura da M++ em grande detalhe, e foi explicado como a especificação foi implementada em Verilog. Por fim, foram apresentados os programas e ferramentas utilizados, bem como os resultados alcançados pelo desenvolvimento.

4.1 REQUISITOS

A aplicação deste trabalho deve:

a) armazenar programa em memória interna (Requisito Funcional – RF);
b) ler programa armazenado em memória interna (RF);
c) ler entradas acionadas pelo usuário (RF);
d) acionar saídas conforme processamento do programa (RF);
e) ser programado na linguagem de descrição de hardware Verilog no ambiente de programação Intel Quartus Prime (Requisito Não Funcional – RNF);
f) ter a funcionalidade testada e comprovada com o software de simulação ModelSim*-Intel® FPGA edition (RNF);
g) ser implementado em um kit de desenvolvimento da Altera, com um FPGA da família Cyclone V (RNF);
h) funcionar em um clock superior à 10KHz (RNF).

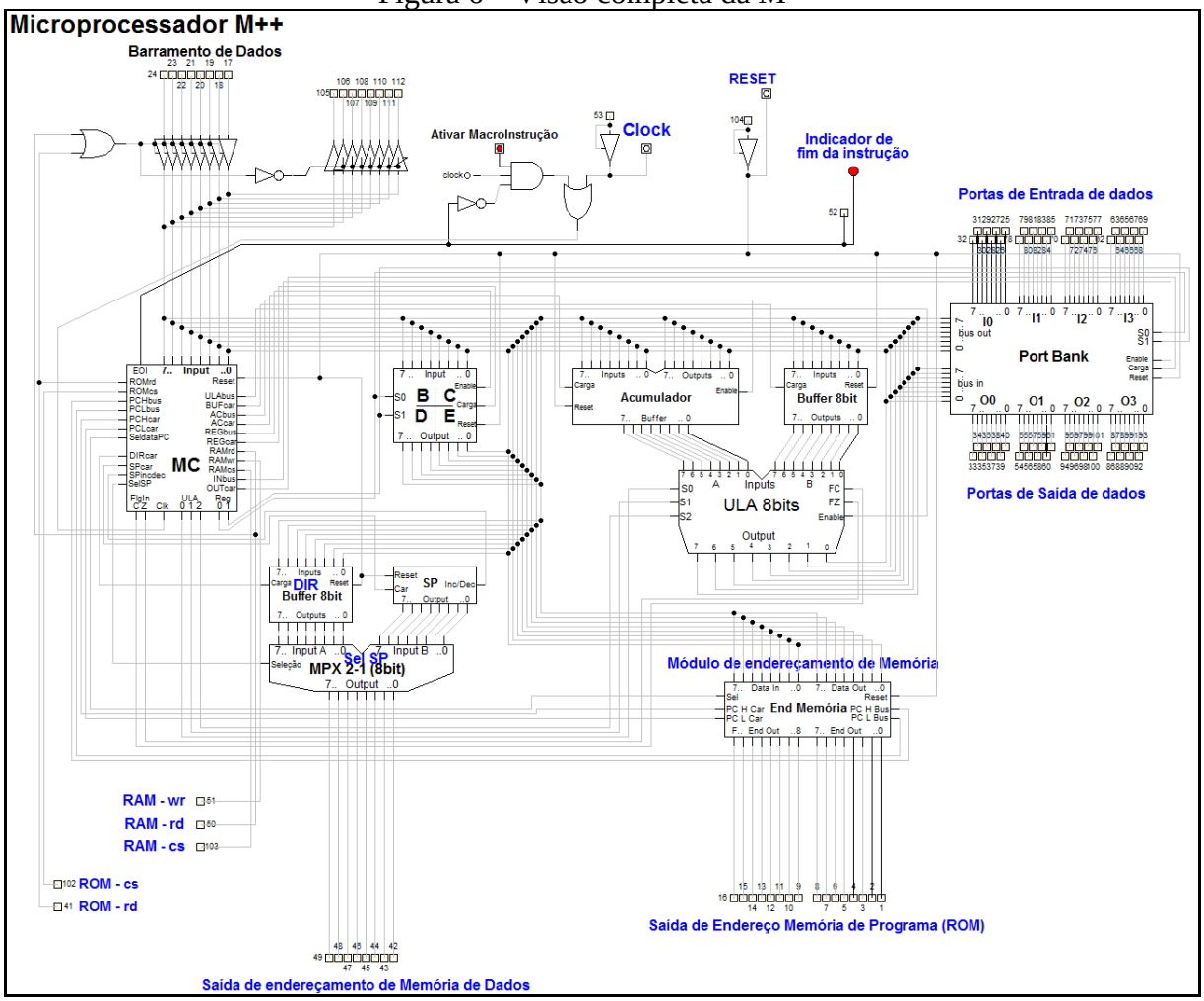
4.2 ESPECIFICAÇÃO

O projeto foi desenvolvido como a implementação da M++ em um FPGA, e por isso a própria M++ será utilizada como a especificação do trabalho. Nas próximas seções, sua arquitetura será explicada detalhadamente.

4.2.1 M++

O módulo M++ é o de mais alto nível e contém todos os outros módulos, como visto na Figura 6, notando que não há um módulo específico para o endereçamento da memória RAM, mas os componentes que executam esta funcionalidade estão neste módulo, e a explicação da função está em uma seção própria. Há um barramento de 8 bits que conecta todos os módulos, e pode receber dados dos mesmos, mas apenas um módulo pode atribuir um valor ao barramento por vez, para não haver curtos circuitos.

Figura 6 – Visão completa da M++



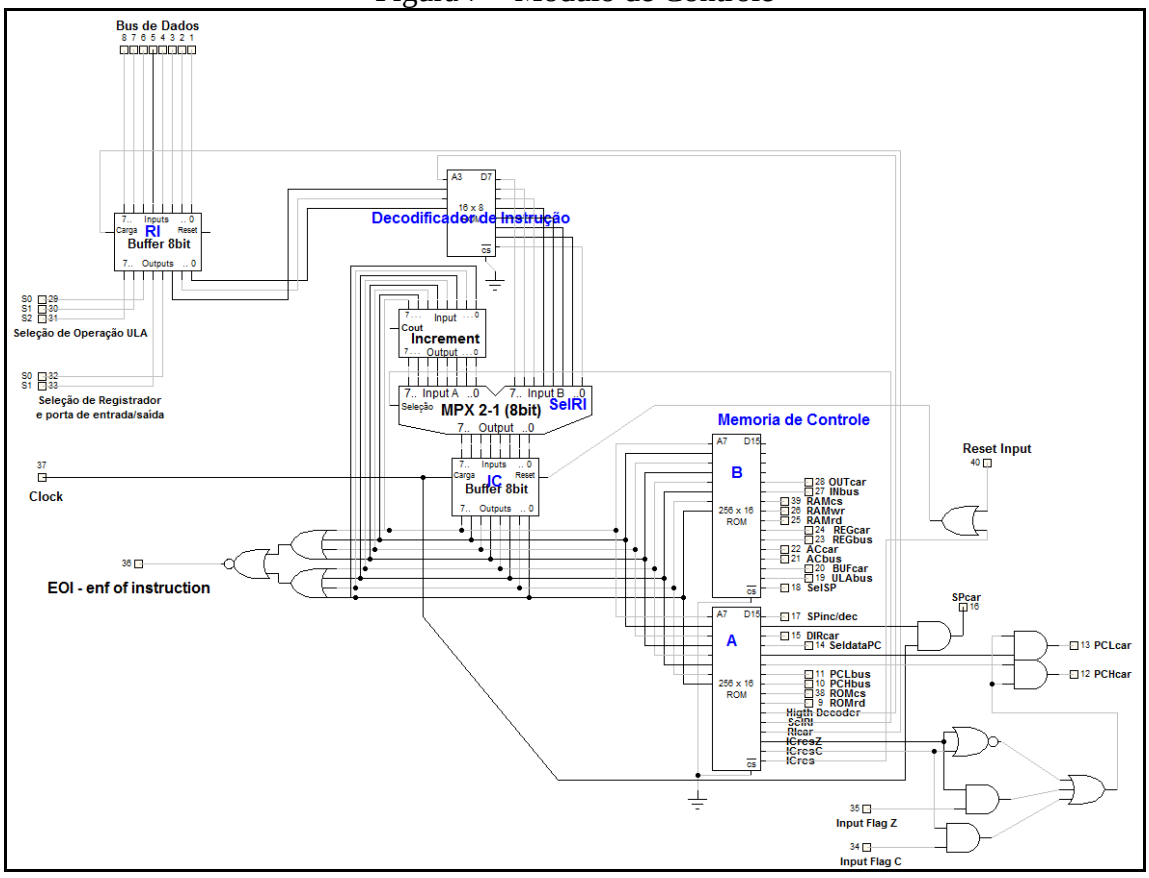
Fonte: Borges (2003).

4.2.2 Módulo de Controle

É a parte principal do processador e comanda todos os outros módulos através dos sinais de saída de duas memórias ROM, chamadas memórias de controle. São duas memórias ROM com 256 posições de memória e dados de 16 bits, totalizando 32 bits. Conforme as memórias recebem novos endereços, os sinais de saída comandam as outras partes do processador. Cada instrução realizada pelo processador é uma sequência de posições das memórias de controle, que executam vários comandos, e essa sequência é chamada de fluxo de operação. Algumas instruções tem um fluxo de operação pequeno, enquanto outras instruções mais complexas, precisam executar mais passos, tem um fluxo de operação maior. Sempre que o sinal `clk` estiver na borda de subida, este acionará o buffer IC, que lerá um novo comando do fluxo, ou seja, enquanto o sinal do `clk` estiver oscilando, novos comandos serão executados em toda borda de subida.

A Figura 7 mostra o circuito responsável pelo controle dos fluxos. O multiplexador `SelRI` define se o fluxo atual continuará pelo pino, quando o valor o pino de seleção é 0, para selecionar a entrada A, ou se será iniciado um novo fluxo, que ocorre com o pino de seleção em 1, e seleciona a entrada B. A saída do multiplexador está conectada à entrada do *buffer* IC, ou seja, na próxima borda de subida do `clk`, o endereço das memórias de controle será o valor atual da saída do multiplexador. A entrada A do multiplexador `SelRI` está conectada à saída de um incrementador, cuja entrada recebe o valor do endereço atual das memórias, e sempre atribui à saída o valor de entrada mais 1, fazendo com que o próximo comando seja sempre o próximo ao anterior. A entrada B do multiplexador `SelRI` está conectada à memória de decodificação de instrução. Quando uma instrução chega ao final, o último comando é sempre acionar o sinal `ICres`, bit número 0 da memória de controle A, que zera o valor do *buffer* IC e dá início ao fluxo de operação de ler uma nova instrução da memória do programa. Quando é terminada a leitura da nova instrução, é selecionada a entrada B do multiplexador com valor da nova instrução.

Figura 7 – Módulo de Controle



Fonte: Borges (2003).

Os próprios sinais da memória de controle comandam esta escolha. O pino de seleção do multiplexador é o sinal `SelRI`, que vem do *bit* número 4 da memória de controle A. O

padrão é receber o valor incrementado como novo endereço e continuar a executar a instrução atual, até que o sinal determine o início de um novo fluxo de operação, recebido pela memória de decodificação.

A memória de decodificação tem 16 posições, e possui como valores a posição de início dos fluxos de operação de todas as instruções disponíveis. Os *bits* 0-2 do endereço da memória vêm da instrução, e o quarto bit do endereço vem de um dos sinais da memória de controle, chamado de `High Decoder`. Como o barramento tem apenas 8bits, e há apenas 3 bits para o endereço da memória de decodificação, foi criado o sinal `High Decoder` para que haja mais 8 instruções. Caso não houvesse o sinal de `High Decoder`, haveria apenas 8 instruções, e as funcionalidades da arquitetura seriam bastante limitadas. O valor da instrução para chamar o `High Decoder` é `0x07`, e o fluxo de operação desta instrução é carregar a próxima instrução, e colocar o valor do `High Decoder` em nível alto, fazendo com que a próxima instrução acesse os endereços 8-15 da memória de decodificação. As instruções que não são `0x07` executam seu fluxo de operação normalmente, e a memória de decodificação acessa as posições 0-6.

Para a recepção de uma nova instrução há o *buffer* `RI`, cuja entrada está ligada ao barramento. O *buffer* apenas carrega uma nova instrução para a memória de decodificação quando recebe o sinal `RIcar`, que salva o valor da entrada no momento em que o sinal esteve na borda de subida. Os *bits* 0-2 são utilizados como endereços na memória que decodifica as instruções, os *bits* 3-4 definem em qual registrador do banco de registradores será realizada a operação, e os *bits* 5-7 definem qual operação da ULA será realizada.

É interessante notar como a escolha de divisão dos *bits* foi útil junto com o `High Decoder`. Por exemplo, caso não houvesse o `High Decoder` e fosse necessário mais um *bit* para ter mais instruções, seria necessário remover este *bit* ou da seleção dos registradores, possibilitando ter apenas um registrador, ou da ULA, reduzindo o número de operações possíveis pela metade, e em ambos os casos a funcionalidade da arquitetura seria muito mais limitada. Com o sinal de `High Decoder` foi possível estender a capacidade do processador com pouco custo de complexidade.

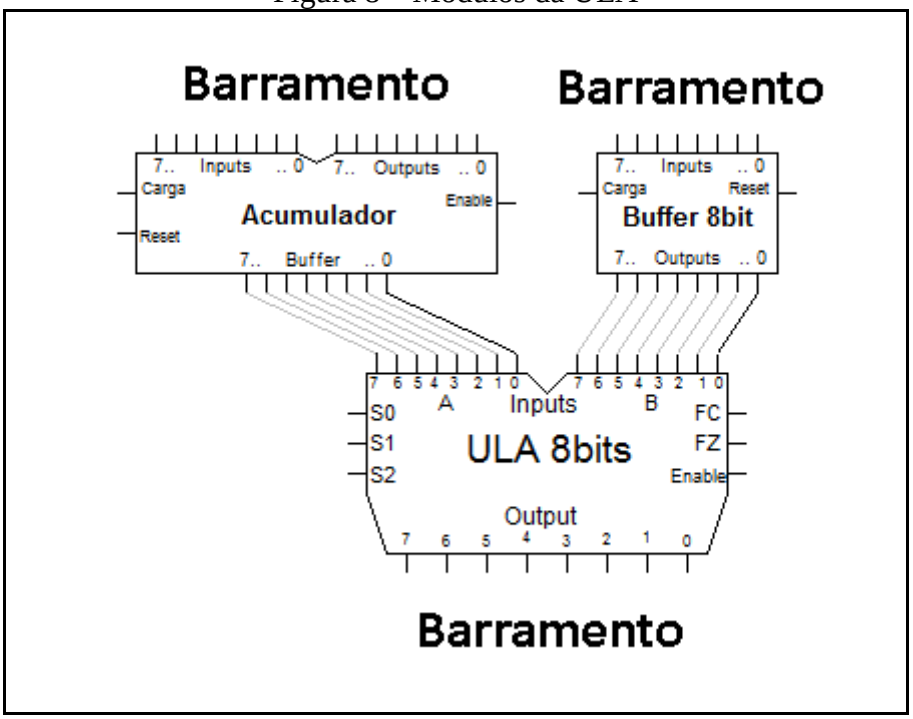
O fluxo de operação mais executado é o de buscar uma nova instrução na memória do programa, e é executado quando o endereço das memórias de controle é zero. Os comandos que ocorrem em seguida são: carregar uma nova instrução da memória do programa, decodifica-la e inicializar o respectivo fluxo de operação. O comando `ADD`, por exemplo, primeiro carrega um valor no acumulador, outro no *buffer* da ULA, executa a operação e

armazena o resultado no acumulador. Em seguida ele zera o valor armazenado no *buffer* IC e são iniciados os comandos para carregar uma nova instrução.

4.2.3 ULA

É o módulo responsável pelas operações lógicas e aritméticas. As operações disponíveis são: soma, subtração, `AND`, `OR`, `XOR`, `NOT`, movimentação e incremento. Conforma a Figura 8 Fazem parte do conjunto da ULA também o acumulador, um registrador intermediário para armazenar valores temporários para serem usados em alguma operação e o *buffer*, utilizado para receber o dado no qual a operação será realizada sobre. Para somar dois valores, é necessário mover um valor para o acumulador e então chamar a instrução `ADD B,A;`, por exemplo, que soma o valor de B com o valor armazenado no acumulador e armazena o resultado no acumulador.

Figura 8 – Módulos da ULA



Fonte: Borges (2003).

As portas de entrada da ULA são as seguintes: 3 *bits* para a seleção da operação, `S0`, `S1` e `S2`, conforme o Quadro 10, a entrada A de 8 *bits* que vem do acumulador, a entrada B que vem do *buffer* e do sinal `enable`. O resultado da operação selecionada fica em `tri-state` e o sinal `enable` habilita o *tri-state* e joga o resultado da operação selecionada no barramento.

Quadro 10 – Operações conforme sinais de seleção

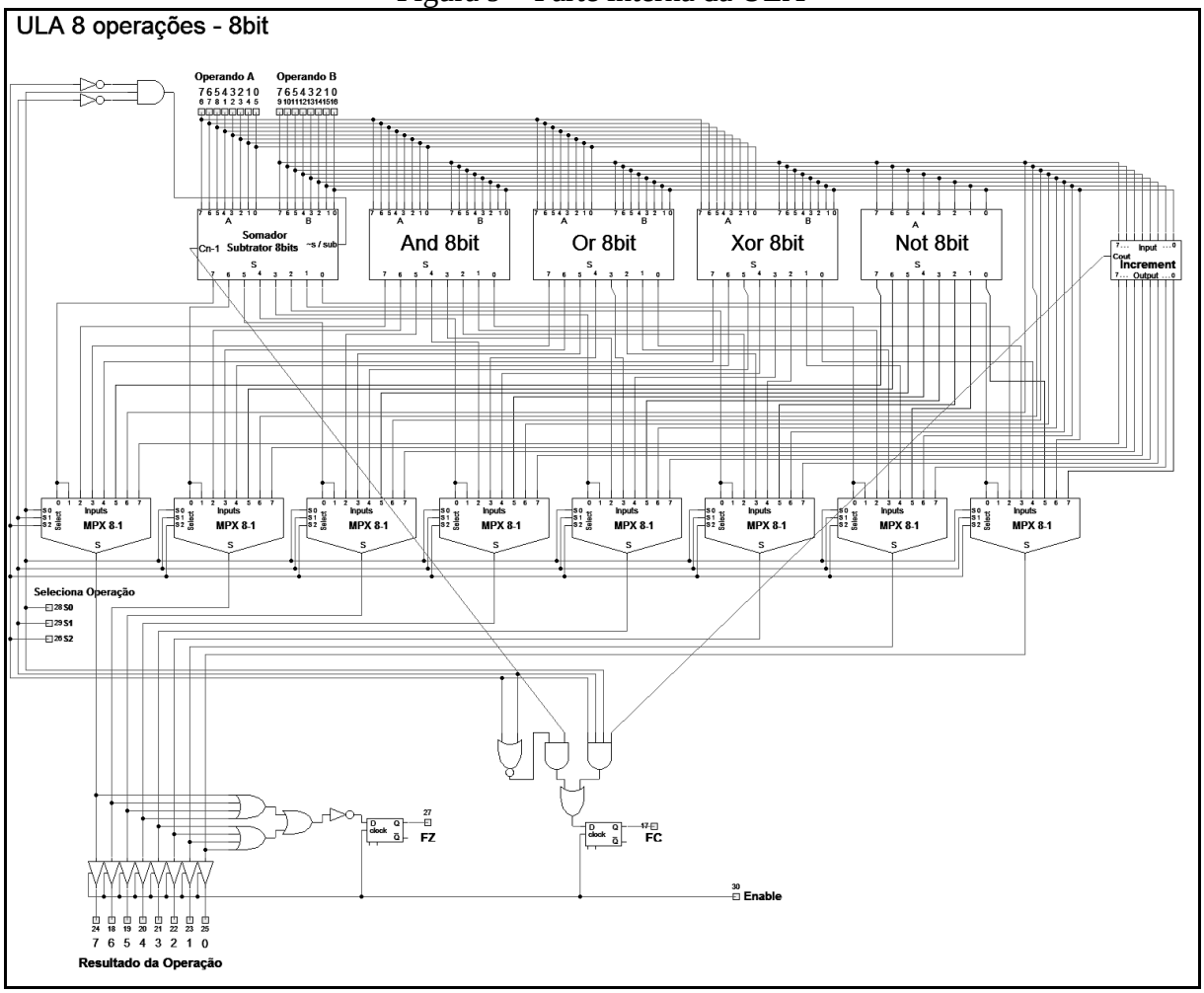
S2	S1	S0	Operação
0	0	0	Soma
0	0	1	Subtração
0	1	0	E
0	1	1	Ou
1	0	0	Ou exclusivo
1	0	1	Negação
1	1	0	Movimentação
1	1	1	Incremento

Fonte: elaborado pelo autor.

As portas de saída da ULA são: saída de 8 *bits*, que é o resultado da operação em tri-state, e duas *flags* de resultados especiais, sendo a FC a flag que indica o *overflow* nas operações de soma e incremento e a FZ a *flag* que indica que o resultado de alguma operação foi zero. As duas *flags* são usadas para os comandos de JMP condicionais JMPC e JMPZ.

Dentro da ULA há blocos que executam cada operação, conforme visto na Figura 9. A entrada de cada bloco de operação está ligada às entradas e os blocos sempre calculam o resultado independente da operação selecionada. Há 8 multiplexadores 8-1, ou seja, 8 bits de entrada e 1 bit de saída, que são responsáveis por colocar o resultado correto na saída. Cada um dos bits de entrada está ligada à uma das operações, de forma que quando a operação de movimentação é selecionada, por exemplo, todos os multiplexadores irão atribuir o *bit* número 6 da entrada à saída. As saídas dos 8 multiplexadores formam os 8 bits de saída.

Figura 9 – Parte interna da ULA

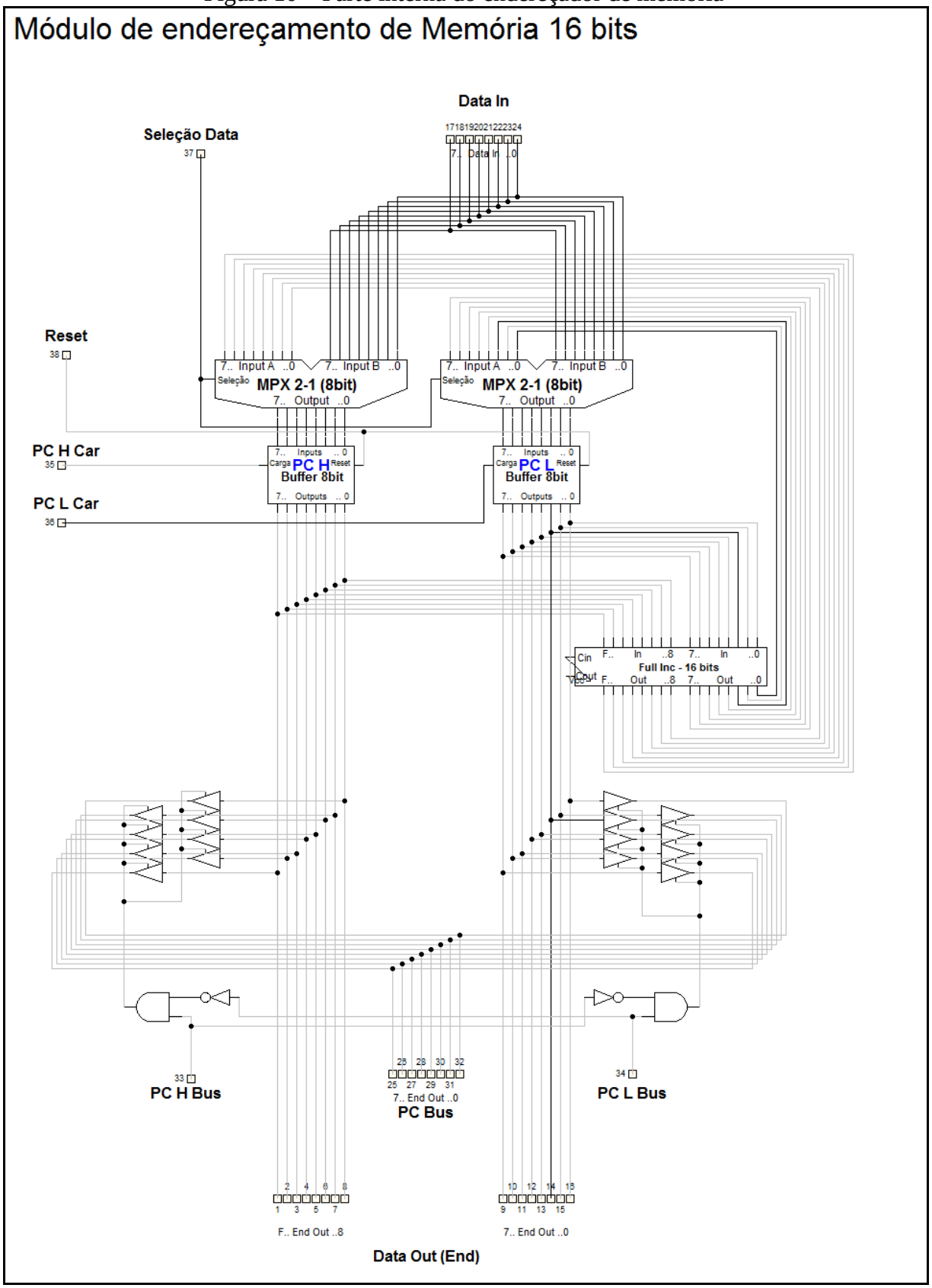


Fonte: Borges (2003).

4.2.4 Endereçador da Memória de Programa

Serve para controlar o endereço da memória do programa, e sua parte interna é exibida na Figura 10. Como a mesma tem um endereço de 16 *bits*, pode se ter um programa com até 65536 instruções. Foram utilizados 2 circuitos de 8 *bits* para o endereçamento, sendo que um controla os 8 *bits* menos significativos e o outro controla os 8 *bits* mais significativos. Os *buffers* `PCL` e `PCH` armazenam o endereço atual, para o *byte* menos significativo e para o mais significativo, respectivamente. Cada *buffer* tem sua entrada conectada a um multiplexador 2-1, com duas entradas e 8 *bits* e uma saída de 8 *bits*. Os multiplexadores controlam qual será a próxima instrução, sendo que a entrada A recebe o valor atual do endereço somado 1, gerado por um incrementador, e define que a próxima instrução é a sequência da anterior. A entrada B recebe o valor do barramento, e é utilizada nas instruções `JMP` e suas variações condicionais, `CALL` e `RET`, quando é necessário ir para outro endereço qualquer da memória. O sinal `Sel` está ligado ao pino de seleção dos multiplexadores.

Figura 10 – Parte interna do endereçador de memória



Fonte: Borges (2003).

O motivo para dividir em dois circuitos é que há a possibilidade de receber um novo endereço de programa, pelas instruções de `JMP`, `CALL` e `RET`, pelo barramento, que tem apenas 8 *bits*. Por isso, na hora de receber uma nova instrução, há um comando para receber os 8 *bits* mais significativos, e outro comando para os 8 *bits* menos significativos. A instrução `JMP` tem o seguinte formato:

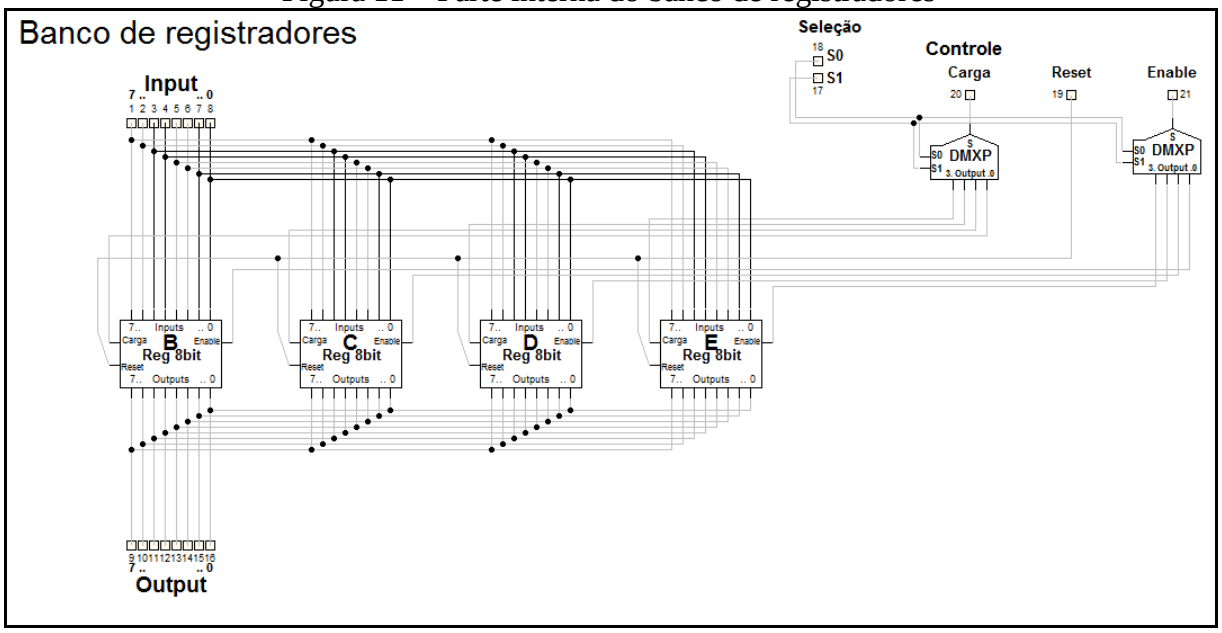
1. 0x07 - Chama o `High Decoder` pois o `JMP` é uma instrução que pertence aos *bits* 8-15 da memória de decodificação;
2. 0x03 - Chama a instrução `JMP`
3. 0x00 - 8 *bits* mais significativos do novo endereço
4. 0x00 - 8 *bits* menos significativos do novo endereço

Nas instruções `CALL` e `RET`, além de receber um novo endereço, é preciso também salvar o endereço atual para que o programa possa retornar a ele quando a instrução RET for chamada. Para isto, há os sinais `PCLbus` e `PCHbus`, que atribuem o valor atual do endereço ao barramento, sendo o `PCLbus` para o *byte* menos significativo e o `PCHbus` para o *byte* mais significativo. A saída está sempre em *tri-state*, para evitar curto circuito quando qualquer outro módulo estiver atribuindo algum valor ao barramento, e há duas portas `AND` que bloqueiam a saída para o barramento caso ambos os sinais estejam ativos ao mesmo tempo.

4.2.5 Banco de Registradores

Um vetor de quatro registradores internos para armazenar dados dentro do processador, cuja parte interna é exibida na Figura 11. Há dois sinais de seleção, `S0` e `S1`, para escolher em qual dos registradores será feita a operação. A entrada de dados de 8 *bits* está ligada à entrada de todos os registradores, mas o sinal de carga que salva o dado da entrada é acionado para apenas um registrador, por causa do demultiplexador `Carga`. Assim como a entrada, a saída para o barramento também é compartilhada para todos os registradores, mas o sinal `enable`, que coloca o dado armazenado para a saída, é apenas acionado para um registrador, por causa do demultiplexador `Enable`. Desse modo não há chances de curtos pois a saída de todos os registradores possui *tri-state*. O sinal de reset zera o valor de todos os registradores.

Figura 11 – Parte interna do banco de registradores



Fonte: Borges (2003).

4.2.6 Endereçador da Memória RAM externa

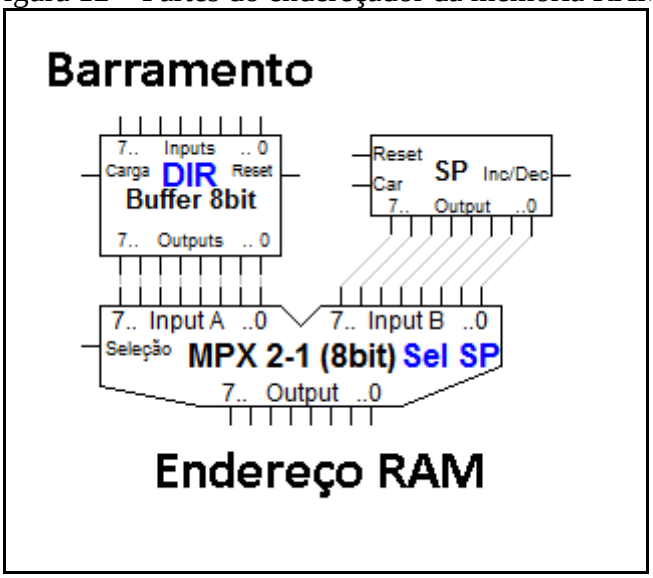
Fazem parte do endereçador o *buffer* `DIR`, um contador de incremento e decremento de 8 *bits*, chamado `SP`, e um multiplexador 2-1 de 8 *bits*, como mostra a Figura 12. A memória é utilizada para armazenar dados da pilha de chamadas e para armazenar dados do usuário. O *buffer* `DIR` é utilizado para endereçar alguma posição definida pelo usuário. Sua entrada está conectada ao barramento e salva o valor atual quando o sinal `DIRcar` está na borda de subida. O contador `SP` é utilizado para armazenar endereços na pilha (ver seção 4.2.7). O sinal `SPincdec` define se o valor será incrementado ou decrementado. Quando o valor é 0, a saída será igual à contagem atual mais 1, e quando o valor for 1, a saída será igual à contagem atual menos 1. O sinal `SPcar` é o sinal de carga do contador. Sempre que estiver na borda de subida, o contador irá executar a operação definida pela entrada `SPincdec`. Pelo sinal `SelSP`, o multiplexador `Sel SP` define se o endereço da RAM virá da entrada A, conectada à saída do *buffer* `DIR`, ou da entrada B, conectada à saída do contador. O sinal de *reset* zera tanto o contador quanto o *buffer*.

Há dois casos para armazenar um dado na pilha: a instrução `JMP` e suas variações condicionais e a `CALL/RET`. O `JMP` irá gravar o endereço que o programa irá "pular" para na pilha, e o `CALL/RET` primeiro irá gravar o endereço atual do programa, e depois irá empilhar o endereço que está sendo "chamado".

Como a memória do programa tem um endereço de 16 *bits*, e a memória RAM guarda apenas dados de 8 *bits*, sempre são salvos 2 *bytes* de memória para cada endereço. O primeiro

byte a ser salvo é o mais significativo e o segundo *byte* é o menos significativo. A pilha começa a ser salva no final da memória RAM, na posição `0xFF` e `0xFE` para a primeira posição, `0xFD` e `0xFC` para a segunda posição e assim por diante.

Figura 12 – Partes do endereçador da memória RAM



Fonte: Borges (2003).

4.2.7 Call Stack

Sempre que os comandos `JMP`, `CALL` e `RET` são chamados, é utilizada uma pilha de endereços para que o programa saiba para que posição da memória do programa pular, e para qual posição retornar, no caso do comando `RET`. O Quadro 11 mostra um programa que utiliza o `JMP` para retornar ao início do programa. Quando a instrução de `JMP` está sendo processada, o endereço, neste caso 0, definido pelo *label* `loop`, em que o programa deve realizar o `JMP` é salvo na primeira posição da pilha.

Quadro 11 – Programa com `JMP` para o início

```
loop:
MOV 44,A;
MOV A,B;

MOV A,OUT1;

JMP loop;
```

Fonte: Borges (2003).

O Quadro 12 mostra o código de um programa que utiliza as instruções `CALL` e `RET`, que são utilizadas para chamar sub-rotinas e retornar à posição de execução anterior. Neste caso, primeiro é adicionada a posição atual do programa, para que o programa saiba para qual instrução retornar quando executar a instrução RET, e em seguida é adicionada a posição que será realizada a CALL.

Quadro 12 – Programa com `CALL` e `RET`

```
loop:
MOV 55,A;
MOV 66,B;
CALL SOMA;
JMP LOOP;

SOMA:
ADD A,B;
RET;
```

Fonte: BORGES (2003).

4.3 IMPLEMENTAÇÃO

Esta seção tem por objetivo mostrar como foi feita a implementação em Verilog para cada funcionalidade descrita na arquitetura. Para quase todo bloco da M++, foi criado um módulo em Verilog para representá-lo.

Para a implementação foi utilizada a linguagem de descrição de hardware Verilog, na versão Verilog-2001, definida pela norma IEEE 1364-2001 (IEEE, 2001). A IDE Quartus foi utilizada para a implementação e compilação do código e o programa ModelSim*-Intel® FPGA edition para a simulação do design. O sistema de controle de versão Git foi utilizado para controlar as modificações feitas a cada versão nova do projeto. Para facilitar o acesso das funções do Git, foi utilizado o software TortoiseGit, que adiciona as funções do Git ao menu de contexto do Windows. O repositório do projeto pode ser acessado em: https://github.com/bieging/mpp.

A M++ foi implementada usando a metodologia *top-bottom*, na qual o módulo `mpp` foi o primeiro a ser definido. Este foi dividido nos sub-blocos necessários, que foram criados em seguida.

Para melhor entendimento durante a explicação dos módulos, algumas partes do código foram cortadas, para evitar poluição visual de código não relacionado ao contexto da explicação. Como é uma tecnologia pouco utilizada no curso, será explicada a implementação de cada parte, pois todas são consideradas importantes.

4.3.1.1 M++

O módulo `ctrl_module` é o módulo principal e contém todos os outros módulos. Sua principal função é conectar os módulos pelo barramento e distribuir os sinais de controle do módulo de controle.

A instanciação de alguns módulos é mostrada no Quadro 13. Algo que é interessante no caso de entradas de módulos que não precisam de tratamento ou verificação, é que é possível conectar a saída de alguns módulos com a entrada de outros. Não é necessário fazer a

recepção desta variável através de uma variável do tipo `wire` e então usar esta variável como a entrada do outro módulo. Criando-os desta forma, já faz com que o sinal de saída esteja conectado diretamente com a entrada do outro módulo. No módulo `program_addresser` foi criado um vetor de sinais de controle de entrada, para não haver uma lista de terminais muito grande. A criação do vetor também foi feita para o banco de registradores e para a ULA nos sinais de seleção de registrador e operação, respectivamente. No caso do banco de registradores há a facilidade de indexar o vetor de registradores utilizando o vetor de seleção, e no caso da ULA, pode-se fazer uma expressão `case` com o vetor de seleção.

Quadro 13 – Instanciação dos módulos

```
ctrl_module cm
    (
        .clk            (clk),
        .instruction    (instruction),
        .ctrl_signals   (ctrl_signals),
        .sp_car         (sp_car)
    );

program_addresser pa
    (
        .ctrl_signals   (program_signals),
        .reset          (reset),
        .in             (bus),
        .out_low        (program_low),
        .out_high       (program_high)
    );

reg_bank rb
    (
        .sel    (reg_bank_sel),
        .en     (ctrl_signals[22]),
        .load   (ctrl_signals[21]),
        .reset  (reset),
        .in     (bus),
        .out    (reg_bank_out)
    );

accumulator acc
    (
        .en     (ctrl_signals[24]),
        .load   (ctrl_signals[23]),
        .reset  (reset),
        .in     (bus),
        .out    (acc_out),
        .buffer (acc_buffer)
    );

alu al
    (
        .sel    (alu_sel),
        .en     (ctrl_signals[26]),
        .in_a   (acc_buffer),
        .in_b   (alu_buffer),
        .flag_c (alu_fc),
        .flag_z (alu_fz),
        .out    (alu_out)
    );
```

Fonte: elaborado pelo autor.

A maioria das variáveis neste módulo foram criadas para as entradas e saídas dos outros módulos, cujos sinais não são conectados diretamente. Variáveis do tipo `reg` são criadas para a entrada e variáveis do tipo `wire` para a saída. Outras variáveis criadas são relacionadas ao endereçamento da memória RAM, que tem toda sua lógica neste módulo, porém sua explicação é na seção 4.3.1.6. A variável mais importante é a `bus`, que representa o barramento, e é responsável pela transmissão dos dados entre todos os módulos. Pode ser visto que todos os módulos que recebem algum dado têm a variável `bus` como entrada, enquanto os que produzem algum valor atribuem este valor ao barramento quando solicitado, conforme visto no Quadro 14.

Quadro 14 – Bloco `always` de atribuição ao barramento

```
always @ (*)
    begin
        if (out_signals[0] == 1'b1)
            begin
                bus = instruction;
            end
        else if (storage_cs == 1'b1)
            begin
                bus = storage_data_out;
            end
        else if (ctrl_signals[2] == 1'b1)
            begin
                bus = program_high;
            end
        else if (ctrl_signals[3] == 1'b1)
            begin
                bus = program_low;
            end
        else if (ctrl_signals[22] == 1'b1)
            begin
                #5
                bus = reg_bank_out;
            end
        else if (ctrl_signals[24] == 1'b1)
            begin
                #5
                bus = acc_out;
            end
        else if (ctrl_signals[26] == 1'b1)
            begin
                #5
                bus = alu_out;
            end
        else if (ctrl_signals[17] == 1'b1)
            begin
                bus = in;
            end
    end
```

Fonte: elaborado pelo autor.

Há três condições em que há um atraso de 5 unidades de tempo antes que o barramento receba o novo valor, para evitar que haja o *Race Condition*. Na primeira condição com *delay*, na qual o barramento recebe o valor de saída do banco de registradores, há um atraso pois o sinal `ctrl_signals[22]`, que atribui ao barramento o valor de saída do banco de

registradores, é o mesmo que habilita a saída do banco de registradores. Como ambas as funções são executadas no mesmo instante, pode ser que quando o valor for atribuído ao barramento, este ainda não tenha sido atualizado pelo banco de registradores, colocando o valor errado na variável `bus`. Na segunda condição, na qual o barramento recebe o valor de saída do acumulador, o Race Condition acontece pelo mesmo motivo da primeira, pois o sinal `ctrl_signals[24]` é o mesmo que atribuí a saída do acumulador ao barramento e que habilita a saída no acumulador, não garantindo que o valor esteja correto quando o barramento for receber o novo valor. A última, que atribui o valor da ULA ao barramento, possui um atraso pelo mesmo motivo, e não há garantia do valor correto na atribuição da saída ao barramento. Ao adicionar um atraso, há certeza de que o valor atribuído ao barramento estará com o valor correto.

No bloco `always @ (ctrl_signals)`, há o preenchimento dos vetores de comando do módulo de estímulo e do módulo de endereçamento do programa, dos vetores de seleção do banco de registradores e da ULA e também há a lógica do endereçador da memória RAM, que é explicado na seção 4.3.1.6. Os blocos `always @ (program_low)` e `always @ (program_high)` apenas recebem um novo endereço do endereçador da memória do programa e o atribuem ao vetor de endereço de saída, que é recebido no módulo de estímulo. O bloco `always @ (posedge ctrl_signals[25])` serve apenas como o sinal de *enable* do buffer da ULA.

4.3.1.2 Módulo de Controle

Foi criado o módulo `ctrl_module` para o Módulo de Controle. Há 3 módulos filhos, que são a memória de decodificação, a memória de controle A e a memória de controle B. O resto da lógica foi implementada no próprio módulo.

O Quadro 15 mostra como é feita a recepção de uma nova instrução e a divisão dos 8 *bits* para cada função. Sempre que o valor da variável `ctrl_a_data[3]` muda, os 3 primeiros *bits* da instrução são atribuídos aos 3 primeiros *bits* do endereço da memória de decodificação. Os outros 5 *bits* são atribuídos às posições 11-15 do vetor `ctrl_signals`, sendo 2 *bits* para a seleção no banco de registradores e os outros 3 bits para a seleção de operação na ULA. O quarto *bit* do endereço da memória de decodificação, utilizado para o `High Decoder`, é atribuído pelo sexto *bit* da saída da memória de controle A, representado pela variável `ctrl_a_data[5]`, no bloco `always` indicado pelo Quadro 19.

Quadro 15 – Recepção da instrução

```
always @ (posedge ctrl_a_data[3])
    begin
        decode_address[0] = instruction[0];
        decode_address[1] = instruction[1];
        decode_address[2] = instruction[2];

        ctrl_signals[14] = instruction[3]; // REG0
        ctrl_signals[15] = instruction[4]; // REG1

        ctrl_signals[11] = instruction[5]; // ULA0
        ctrl_signals[12] = instruction[6]; // ULA1
        ctrl_signals[13] = instruction[7]; // ULA2
    end
```

Fonte: elaborado pelo autor.

O Quadro 16 mostra como o `clk` aciona o buffer `IC` para armazenar o valor atual de saída do multiplexador `SelRI`, indicado pela variável `decode_reg`, ou zera o endereço, iniciando o fluxo de operação de leitura de uma nova instrução da memória do programa. Sempre que o `clk` estiver na borda de subida, o programa irá verificar se a variável `ctrl_signals[0]` está em nível alto, para então zerar o valor armazenado. Caso esteja em nível baixo, o *buffer* armazena o valor atual da entrada.

Quadro 16 – Acionamento do *buffer* de endereço das memórias de controle

```
always @ (posedge clk)
    begin
        // Instruction buffer reset.
        if (ctrl_a_data[0] == 1'b1)
            begin
                ctrl_addr = 8'b00000000;
            end
        else
            begin
                ctrl_addr <= decode_reg;
            end
    end
```

Fonte: elaborado pelo autor.

No Quadro 17, o sinal `sp_car` é acionado quando a variável `ctrl_signals[14]` ou o `clk` mudam de estado. Há um *delay* de 5 unidades de tempo antes da execução da instrução, para evitar o *Race Condition*. Como é acionado na mudança de valor das duas variáveis, quando uma muda, a outra pode estar em algum valor indeterminado, gerando resultados errados. Pode-se perceber que o `sp_car` é um sinal de controle, mas foi criado como uma variável separada do vetor `ctrl_signals`, que se deve ao fato de que dois blocos `always` com listas de sensibilidades diferentes não podem atribuir valores para uma mesma variável. Como a maioria das posições do vetor `ctrl_signals` já são atribuídas em outro bloco `always`, atribuir alguma posição do vetor `ctrl_signals` em um bloco `always` com a lista de sensibilidades diferente gera resultados inesperados.

Quadro 17 – Tratamento do sinal `sp_car`

```
always @ (*)
    begin
        #5
        sp_car = ctrl_a_data[14] && clk;
    end
```

Fonte: elaborado pelo autor.

O código no Quadro 18 exerce a função do multiplexador `SelRI`. Quando a entrada A ou a entrada B, representadas pelos vetores `incr_result` e `decode_data`, respectivamente tem algum valor alterado, o bloco é executado. Caso o valor da variável `ctrl_signals[4]` esteja em nível baixo, o vetor `decode_reg` irá receber o valor da entrada A, e caso esteja em nível alto, o vetor `decode_reg` irá receber o valor da entrada B. Este bloco também possui um atraso de 5 unidades de tempo para evitar que haja o *Race Condition*, pois o valor de alguma das entradas pode estar em um valor indeterminado e gerar uma saída errada.

Quadro 18 – Multiplexador `SelRI`

```
always @ (decode_data or ctrl_addr)
    begin
        #5
        if (ctrl_a_data[4] == 1'b0)
            begin
                decode_reg = incr_result;
            end
        else
            begin
                decode_reg = decode_data;
            end
    end
```

Fonte: elaborado pelo autor.

O bloco `always` do Quadro 19 recebe os dados das memórias de controle sempre que estas tiverem uma seus valores alterados. Os sinais `ctrl_signals[4]` e `ctrl_signals[5]` só recebem o valor caso a variável `pclh_condition` esteja em nível alto, o que ocorre somente quando as variáveis `ctrl_a_data[1]` e `ctrl_a_data[2]` estiverem em nível baixo, conforme a porta `NOR` declarada na primeira linha. A variável `ctrl_signals[6]`, que é conectada ao endereçador da memória do programa e que controla seu multiplexador, é atribuída com um atraso de 5 unidades de tempo, para evitar que haja algum valor indeterminado nas suas entradas.

Quadro 19 – Recepção dos sinais das memórias de controle

```
nor u1 (pclh_condition, ctrl_a_data[1], ctrl_a_data[2]);

always @ (ctrl_a_data or ctrl_b_data)
    begin
        // 4th address decoder signal.
        decode_address[3] = ctrl_a_data[5];

        // Control signals.
        ctrl_signals[0] = ctrl_a_data[6]; // ROMrd
        ctrl_signals[1] = ctrl_a_data[7]; // ROMcs
        ctrl_signals[2] = ctrl_a_data[8]; // PCHbus
        ctrl_signals[3] = ctrl_a_data[9]; // PCLbus

        // PCHcar
        if (ctrl_a_data[10] == 1'b1 & pclh_condition == 1'b1)
            begin
                ctrl_signals[4] = 1'b1;
            end
        else
            begin
                ctrl_signals[4] = 1'b0;
            end

        // PCLcar
        if (ctrl_a_data[11] == 1'b1 & pclh_condition == 1'b1)
            begin
                ctrl_signals[5] = 1'b1;
            end
        else
            begin
                ctrl_signals[5] = 1'b0;
            end

        // Reception of all other signals hidden.

        #5 // Signals that have to wait to avoid race condition.

        ctrl_signals[6] = ctrl_a_data[12]; // SelDataPC

    end
```

Fonte: elaborado pelo autor.

4.3.1.3 ULA

Foi criado o módulo `alu` para a ULA. A implementação foi bastante simples, comparado ao design feito no Digital Works, por causa do uso dos operadores lógicos e aritméticos, e também por poder usar a expressão `case`.

Não há nenhuma variável além das definidas na arquitetura. A única diferença é que foi utilizado um vetor para os sinais de seleção de operação, o que possibilita o uso de uma expressão `case` para a seleção de uma operação.

Conforme o código do Quadro 20, sempre que há uma mudança de estado em alguma das variáveis utilizadas, o bloco `always` será acionado. Caso o sinal de *enable* esteja em nível alto, a seleção da operação é feita verificando o valor do vetor `sel` em uma expressão `case`, e a variável `out` recebe o valor determinado pela operação.

Quadro 20 – Bloco *always* que processa a operação da ULA

```
always @ (*)
    begin
        if (en == 1'b1)
            begin
                case (sel)
                    3'b000: out = in_a + in_b;
                    3'b001: out = in_a - in_b;
                    3'b010: out = in_a & in_b;
                    3'b011: out = in_a | in_b;

                    3'b100: out = in_a ^ in_b;
                    3'b101: out = !in_b;
                    3'b110: out = in_b;
                    3'b111: out = in_b + 1;
                endcase
            end
        else
            begin
                out = 8'bzzzzzzzz;
            end
    end
```

Fonte: elaborado pelo autor.

4.3.1.4 Endereçador da Memória de Programa

Foi criado o módulo `program_addresser` para o endereçador da memória de programa. Além das variáveis de entrada e saída definidas na arquitetura, também foram criadas as variáveis `prev_pch_state` e `prev_pcl_state`, para fazer uma lógica de borda de subida, `mid_low` e `mid_high`, os valores de entrada dos buffers e `incr_result` e `incr_result`, entrada e resultado do incrementador, respectivamente. Lembrando que como o endereço é de 16 *bits*, há uma variável para o *byte* mais significativo, definido por "h" ou "*high*" e outra variável para o *byte* menos significativo, definido por "l" ou "*low*".

Para o resultado do incrementador foi utilizado um bloco `assign`. Desta forma, assim que a entrada do incrementador mudar de valor, o resultado já é atribuído à variável. Como não é necessário que esta variável armazene nenhum dado, esta foi criada com o tipo `wire`.

No Quadro 21, sempre que há alguma mudança nos valores do vetor `ctrl_signals`. A expressão `if` faz o papel dos multiplexadores, definindo o valor de entrada dos buffers com base no valor da variável `ctrl_signals[0]`. Em nível baixo, os vetores `mid_low` e `mid_high` recebem o valor da variável `incr_result`, e, em nível alto, recebem o valor de entrada, que é o valor do barramento.

Quadro 21 – Recepção das variáveis `mid_low` e `mid_high`

```
// Assigns.
assign incr_result = incr_input + 1;

// Always.
always @ (ctrl_signals)
    begin
        // Selection (SelDataPC).
        if (ctrl_signals[0] == 1'b0)
            begin
                mid_low[0] = incr_result[0];
                mid_low[1] = incr_result[1];
                mid_low[2] = incr_result[2];
                mid_low[3] = incr_result[3];

                mid_low[4] = incr_result[4];
                mid_low[5] = incr_result[5];
                mid_low[6] = incr_result[6];
                mid_low[7] = incr_result[7];

                mid_high[0] = incr_result[8];
                mid_high[1] = incr_result[9];
                mid_high[2] = incr_result[10];
                mid_high[3] = incr_result[11];

                mid_high[4] = incr_result[12];
                mid_high[5] = incr_result[13];
                mid_high[6] = incr_result[14];
                mid_high[7] = incr_result[15];
            end
        else
            begin
                mid_low[0] = in[0];
                mid_low[1] = in[1];
                mid_low[2] = in[2];
                mid_low[3] = in[3];

                mid_low[4] = in[4];
                mid_low[5] = in[5];
                mid_low[6] = in[6];
                mid_low[7] = in[7];

                mid_high[0] = in[0];
                mid_high[1] = in[1];
                mid_high[2] = in[2];
                mid_high[3] = in[3];

                mid_high[4] = in[4];
                mid_high[5] = in[5];
                mid_high[6] = in[6];
                mid_high[7] = in[7];
            end
    end
```

Fonte: elaborado pelo autor.

Na parte seguinte do mesmo bloco *always*, como visto no Quadro 22, as variáveis de saída recebem o valor de entrada de seu respectivo *buffer* na borda de subida das variáveis `ctrl_signals[2]` e `ctrl_signals[1]`, para o byte menos significativo e mais significativo, respectivamente. As variáveis `prev_pcl_state` e `prev_pch_state` foram utilizadas para verificar se há a borda de subida, pois estas variáveis armazenam o valor anterior das entradas `ctrl_signals[2]` e `ctrl_signals[1]`, respectivamente. O valor da variável `incr_input` também recebe o mesmo valor de entrada do *buffer*.

Quadro 22 – Atribuição às saídas e atualização do valor de entrada do incrementador

```
// Always.
always @ (ctrl_signals)
    begin
        // Clock L (PCLcar).
        if (ctrl_signals[2] == 1'b1 & prev_pcl_state == 1'b0)
            begin
                out_low = mid_low;

                incr_input[0] = out_low[0];
                incr_input[1] = out_low[1];
                incr_input[2] = out_low[2];
                incr_input[3] = out_low[3];

                incr_input[4] = out_low[4];
                incr_input[5] = out_low[5];
                incr_input[6] = out_low[6];
                incr_input[7] = out_low[7];
            end

        prev_pcl_state = ctrl_signals[2];

        // Clock H (PCHcar).
        if (ctrl_signals[1] == 1'b1 & prev_pch_state == 1'b0)
            begin
                out_low = mid_low;

                incr_input[8] = out_high[0];
                incr_input[9] = out_high[1];
                incr_input[10] = out_high[2];
                incr_input[11] = out_high[3];

                incr_input[12] = out_high[4];
                incr_input[13] = out_high[5];
                incr_input[14] = out_high[6];
                incr_input[15] = out_high[7];
            end

        prev_pch_state = ctrl_signals[1];
    end
```

Fonte: elaborado pelo autor.

4.3.1.5 Banco de Registradores

Foi criado o módulo `reg_bank` para o banco de registradores. Como padrão em todos os módulos, no bloco `initial` foram inicializadas todas as variáveis do tipo reg com o valor 0, com a diferença de que neste bloco há o vetor de *bytes* `regs`, que armazena os valores dos registradores. Conforme o código no Quadro 23, o vetor foi inicializado num *loop* `for`, preenchendo cada posição com 0.

Quadro 23 – Inicialização do vetor `regs`

```
reg [7:0] regs [0:3];

integer i;

initial
    begin
        out = 8'b00000000;

        for (i = 0; i < 4; i = i + 1)
            begin
                regs[i] = 8'b00000000;
            end
    end
```

Fonte: elaborado pelo autor.

O Quadro 24 mostra o bloco *always* responsável por todo o funcionamento. Novamente, a possibilidade de trabalhar com vetores simplifica bastante a implementação se comparado ao design feito no Digital Works. O bloco é acionado quando uma das variáveis da lista de sensibilidades muda de valor. Caso a variável `en` esteja em nível alto, a saída receberá o valor do registrador selecionado. Caso a variável `load` esteja em nível alto, o registrador selecionado irá receber o valor atual da entrada. Caso o `reset` esteja em nível alto, um *loop* `for` zera o valor de todos os registradores.

Quadro 24 – Lógica de armazenamento no vetor `regs`

```
always @ (sel or en or load or reset)
    begin
        if (en == 1'b1)
            begin
                out = regs[sel];
            end

        if (load == 1'b1)
            begin
                regs[sel] = in;
            end

        if (reset == 1'b1)
            begin
                for (i = 0; i < 4; i = i + 1)
                    begin
                        regs[i] = 8'b00000000;
                    end
            end
    end
```

Fonte: elaborado pelo autor

4.3.1.6 Endereçador da Memória RAM externa

Não há um módulo específico que contém todo o funcionamento do endereçamento da memória RAM, sendo que toda a lógica e módulos relacionados a esta parte estão no módulo principal `mpp`. Foi criado apenas o contador de incremento e decremento, representado pelo módulo `inc_dec_counter`.

Como é visto no Quadro 25, o módulo de incremento e decremento é bastante simples. No bloco `initial` o valor de saída é zerado. No bloco `always`, que é acionado sempre que a variável `set` está em borda de subida, o valor de saída é incrementado ou decrementado dependendo do valor da variável `ctrl`, sendo 0 para incrementar e 1 para decrementar. Caso o `reset` esteja em nível alto, o valor de saída é zerado.

Quadro 25 – Módulo de incremento e decremento

```
initial
    begin
        out = 8'b00000000;
    end

always @ (posedge set)
    begin
        if (reset == 1'b1)
            begin
                out = 8'b00000000;
            end
        else if (ctrl == 1'b0)
            begin
                out = out + 1;
            end
        else if (ctrl == 1'b1)
            begin
                out = out - 1;
            end
    end
```

Fonte: elaborado pelo autor.

No bloco *always* do Quadro 26, há o restante da lógica que endereça a memória RAM externa. O bloco é acionado sempre que algum valor do vetor `ctrl_signals` é alterado. Há a atribuição da variável `storage_rw`, que define a escrita na memória caso o valor seja 0, e leitura caso o valor seja 1. O *buffer* `DIR` representado pela variável `dir_sp_buffer` recebe o valor do barramento caso a variável `ctrl_signals[7]` esteja em nível alto. O endereço da memória RAM é a saída do multiplexador `Sel SP` definido pela expressão `if`. O buffer está ligado à entrada A do multiplexador, e a saída do contador, ligada à entrada B. Caso o valor da variável `ctrl_signals[10]` esteja em nível baixo, a saída será igual à entrada A, e caso esteja em nível alto, a saída será igual à entrada B. Por fim, há um delay para a atribuição da variável `storage_cs`, para garantir que o valor do sinal `storage_rw` esteja com o valor correto, caso contrário, ocorreria o *Race Condition*. Após outro *delay* de 5 unidades de tempo, o valor de `storage_cs` é forçado a 0, pois caso contrário este seria zerado somente após a mudança de valor do sinal `storage_rw` para 0, o que geraria uma leitura da memória, pois a lista de sensibilidades da memória RAM necessita da mudança de valor apenas do `storage_cs`. Ao ligar com atraso, e desligar antecipadamente, é garantido que quando o `storage_cs` é acionado e desligado, acionando o bloco `always` da memória, o valor do `storage_rw` não tenha mudado.

Quadro 26 – Endereçador da memória RAM

```
always @ (ctrl_signals)
    begin
        storage_rw = ctrl_signals[20]; // RAMrd

        // DIR buffer.
        if (ctrl_signals[7] == 1'b1)
            begin
                dir_sp_buffer = bus;
            end

        // Sel SP MUX.
        if (ctrl_signals[10] == 1'b0)
            begin
                storage_addr = dir_sp_buffer;
            end
        else
            begin
                storage_addr = storage_signals;
            end

        if (ctrl_signals[16] == 1'b1)
            begin
                out <= bus;
            end

        #5
        storage_cs = ctrl_signals[18]; // RAMcs
        #10
        storage_cs = 0; // RAMcs
    end
```

Fonte: elaborado pelo autor.

4.3.1.7 Memórias ROM

São as memórias apenas para leitura. A memória do programa, de decodificação e as duas de controle são todas memórias ROM. Na implementação deste processador foi escolhido para todas as memórias, exceto a de programa, que o sinal de *chip select* estará sempre em nível baixo, ou seja, a memória está sempre habilitada, e quando houver uma mudança de endereço, a saída já terá um novo valor. Para a implementação das memórias, foi criada apenas uma grande expressão `case`, onde cada opção é um endereço da memória e atribui a saída com um valor fixo.

O bloco `always` das memórias de decodificação e de controle são iguais, mudando apenas a quantidade de dados, por isso, por questões de simplicidade, o Quadro 27 mostra o módulo da memória de decodificação, por ter apenas 16 endereços. Assim que o endereço muda, é verificado se o sinal `en` é igual a zero. Como nestas memórias o sinal `en` é fixo em zero, esta expressão será sempre verdadeira. Em seguida, para cada possibilidade do valor do endereço, é atribuído um valor fixo à variável `data`.

Quadro 27 – Memória de decodificação

```
module ram_decode
    (
        en,
        addr,
        data
    );

    input en;
    input [3:0] addr;

    output [7:0] data;

    reg [7:0] data;

    initial
        begin
            data = 8'b00000000;
        end

    always @ (addr)
        begin
            if (en == 1'b0)
                begin
                    case (addr)
                        4'b0000: data = 8'b00000011;
                        4'b0001: data = 8'b00001000;
                        4'b0010: data = 8'b00001101;
                        4'b0011: data = 8'b00010100;

                        4'b0100: data = 8'b00011001;
                        4'b0101: data = 8'b00011110;
                        4'b0110: data = 8'b00100101;
                        4'b0111: data = 8'b00101010;

                        4'b1000: data = 8'b00101100;
                        4'b1001: data = 8'b00110001;
                        4'b1010: data = 8'b00110110;
                        4'b1011: data = 8'b00111101;

                        4'b1100: data = 8'b01000110;
                        4'b1101: data = 8'b01010000;
                        4'b1110: data = 8'b01011001;
                        4'b1111: data = 8'b01101100;
                    endcase
                end
        end

endmodule
```

Fonte: elaborado pelo autor.

O bloco *always* da memória de programa difere apenas na lista de sensibilidades, pois é acionada pelo sinal `program_cs`, conforme o Quadro 28. O sinal `program_cs` desta memória é controlado pelo sinal `ROMcs`, representado pela variável `ctrl_signals[0]` no módulo de controle. Como o barramento de dados é compartilhado com outros módulos, a memória não pode manter a saída sempre ativada, por isso a mesma é habilitada apenas quando há a necessidade de ler uma nova instrução.

Quadro 28 – Memória ROM do programa

```
always @ (program_cs)
    begin
        if (program_cs == 1'b0)
            begin
                case (in_program_addr)
                    'h0000: instruction = 'h07;
                    'h0001: instruction = 'hC0;
                    'h0002: instruction = 'h55;
                    'h0003: instruction = 'h07;
                    'h0004: instruction = 'hC1;
                    'h0005: instruction = 'h66;
                    'h0006: instruction = 'h07;
                    'h0007: instruction = 'h06;
                    'h0008: instruction = 'h00;
                    'h0009: instruction = 'h0E;
                    'h000A: instruction = 'h07;
                    'h000B: instruction = 'h03;
                    'h000C: instruction = 'h00;
                    'h000D: instruction = 'h00;
                    'h000E: instruction = 'h04;
                    'h000F: instruction = 'h07;
                    'h0010: instruction = 'h07;
                endcase
            end
    end
```

Fonte: elaborado pelo autor.

4.3.1.8 Bloco de estímulo / Testbench

Para testar o design, foi criado um bloco de estímulo, chamado `mpp_tb`. Para realizar o teste é necessário apenas gerar um sinal de clk, enviar a instrução vinda da memória do programa, atribuir algum valor à entrada do usuário e verificar se a saída do usuário foi atribuída com o valor esperado. O Quadro 29 mostra a criação das variáveis necessárias para instanciar o módulo `mpp`, e a criação do mesmo.

Quadro 29 – Criação das variáveis

```
module mpp_tb ();

    reg clk;
    reg [7:0] in;

    wire [7:0] out;
    wire [4:0] out_signals;
    wire [15:0] in_program_addr;

    reg [7:0] instruction;

    reg program_cs;

    mpp m1
        (
            .clk          (clk),
            .instruction  (instruction),
            .out_signals  (out_signals),
            .program_addr (in_program_addr),
            .in           (in),
            .out          (out)
        );
```

Fonte: elaborado pelo autor.

No bloco `initial`, visto no Quadro 30, o sinal `clk` é inicializado em zero e é atribuído um valor à variável `in`, que representa a entrada de usuário. Há uma espera de seis mil unidades de tempo antes que seja chamado o comando `$finish`, que solicita a finalização da simulação.

Quadro 30 – Bloco initial

```
initial
    begin
        clk = 1'b0;
        in = 8'b00010100;

        #6000

        $finish;
    end
```

Fonte: elaborado pelo autor.

O Quadro 31 mostra os 3 blocos `always` deste módulo. No primeiro, é gerado o sinal de *clock* para o processador. A cada 10 unidades de tempo, o sinal do `clk` é invertido e aciona todos os módulos inferiores. No próximo `always`, é a recepção do comando de leitura da memória. Como a leitura da memória acontece quando o sinal `program_cs` está em nível baixo, o comando de leitura é sempre invertido, pois vem sempre em nível alto quando é solicitada uma leitura. No último bloco, está a memória do programa. A memória do programa está no bloco de estímulo para evitar a necessidade de recompilar sempre que houver uma alteração no programa, considerando que a compilação do projeto leva de 3 a 5 minutos. Sempre que há uma mudança na variável `program_cs`, é verificado se seu valor é 0, e caso sim, a instrução receberá um novo valor dependendo do endereço.

Quadro 31 – Módulo de estímulo

```
always
    begin
        #10 clk = !clk;
    end

always @ (out_signals)
    begin
        program_cs = ~out_signals[1];
    end

always @ (program_cs)
    begin
        if (program_cs == 1'b0)
            begin
                case (in_program_addr)
                    'h0000: instruction = 'h07;
                    'h0001: instruction = 'hC0;
                    'h0002: instruction = 'h55;
                    'h0003: instruction = 'h07;
                    'h0004: instruction = 'hC1;
                    'h0005: instruction = 'h66;
                    'h0006: instruction = 'h07;
                    'h0007: instruction = 'h06;
                    'h0008: instruction = 'h00;
                    'h0009: instruction = 'h0E;
                    'h000A: instruction = 'h07;
                    'h000B: instruction = 'h03;
                    'h000C: instruction = 'h00;
                    'h000D: instruction = 'h00;
                    'h000E: instruction = 'h04;
                    'h000F: instruction = 'h07;
                    'h0010: instruction = 'h07;
                endcase
            end
    end
```

Fonte: elaborado pelo autor.

4.3.2 Técnicas e ferramentas utilizadas

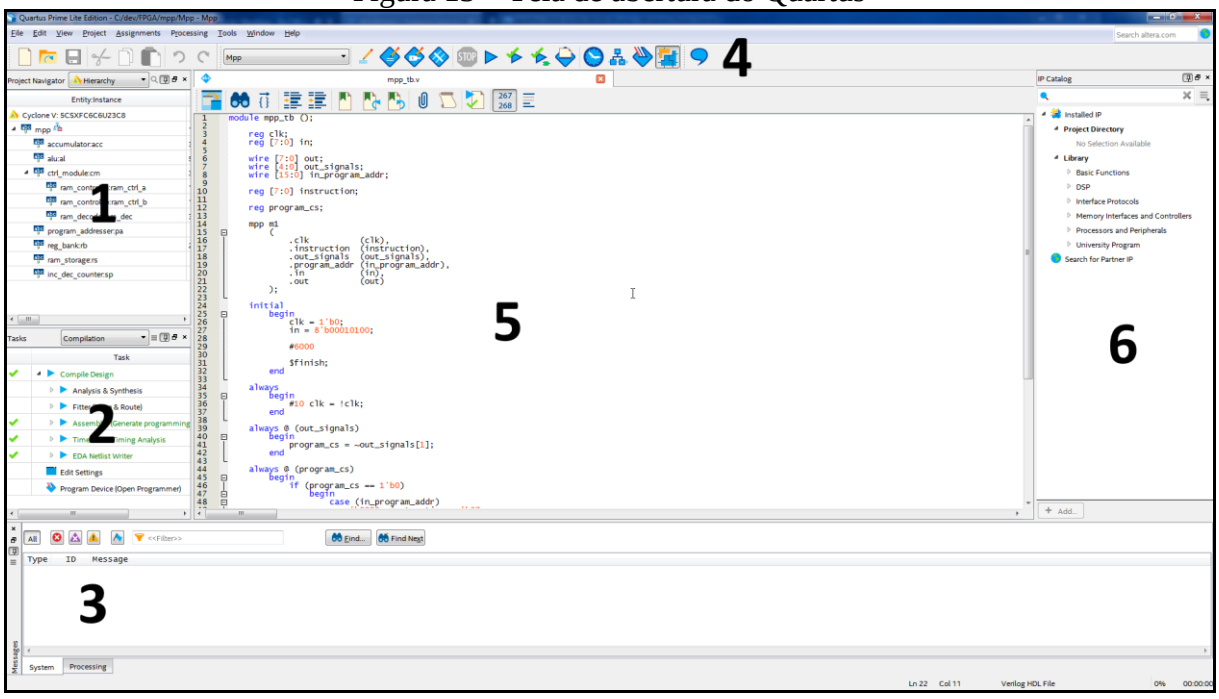
Foram necessários vários programas para o desenvolvimento do projeto. Os softwares principais foram o Quartus, que foi utilizado para o desenvolvimento do código e compilação do design, e o ModelSim*-Intel® FPGA edition, usado para simular o projeto. O programa para criar os programas de teste é o MontadorMmaismais. O software MapReader foi criado para traduzir os arquivos `.map` em código Verilog.

4.3.2.1 Quartus

É o programa fornecido pela Altera para o desenvolvimento em FPGAs. Foi utilizada a edição Lite, que não necessita de licença paga e é destinada ao desenvolvimento das famílias de FPGA de baixo custo, e a versão do programa durante todo o desenvolvimento é a 2017.1. Entre todas as funções, as principais para o desenvolvimento deste projeto são: ambiente de desenvolvimento em Verilog, simulação de tempos, para verificar qual a velocidade máxima suportada pelo design, visualização dos circuitos gerados pela compilação, integração com o software ModelSim*-Intel® FPGA edition para a simulação e programação dos chips.

A Figura 13 mostra a tela de abertura do Quartus. Cada uma das partes principais está indicada por um número. A seção 1 é o navegador do projeto, e mostra a hierarquia dos módulos criados e os dados utilizados por cada módulo. A seção 2 mostra o estado das tarefas de compilação e também, caso haja alguma falha, indica em qual das partes da compilação esta foi detectada. Na seção 3 são exibidas as mensagens geradas pela compilação e também indica caso haja avisos ou erros no código. Caso seja encontrado algum erro, a compilação é interrompida. O menu superior, visto na seção 4, tem diversas opções, sendo as principais utilizadas durante o projeto: compilar, abrir o simulador TimeQuest Timing Analyzer, abrir o relatório de compilação e abrir as configurações do projeto. A área de edição de código é exibida na seção 5. A funcionalidade desta tela é padrão, e há apenas duas funções notáveis, que são: inserir um *template* de alguma funcionalidade já criada e analisar o arquivo atual, para ver se há algum erro sem precisar compilar todo o projeto. O catálogo de propriedades intelectuais, visto na seção 6, fornece diversos módulos complexos já criado pela Intel.

Figura 13 – Tela de abertura do Quartus

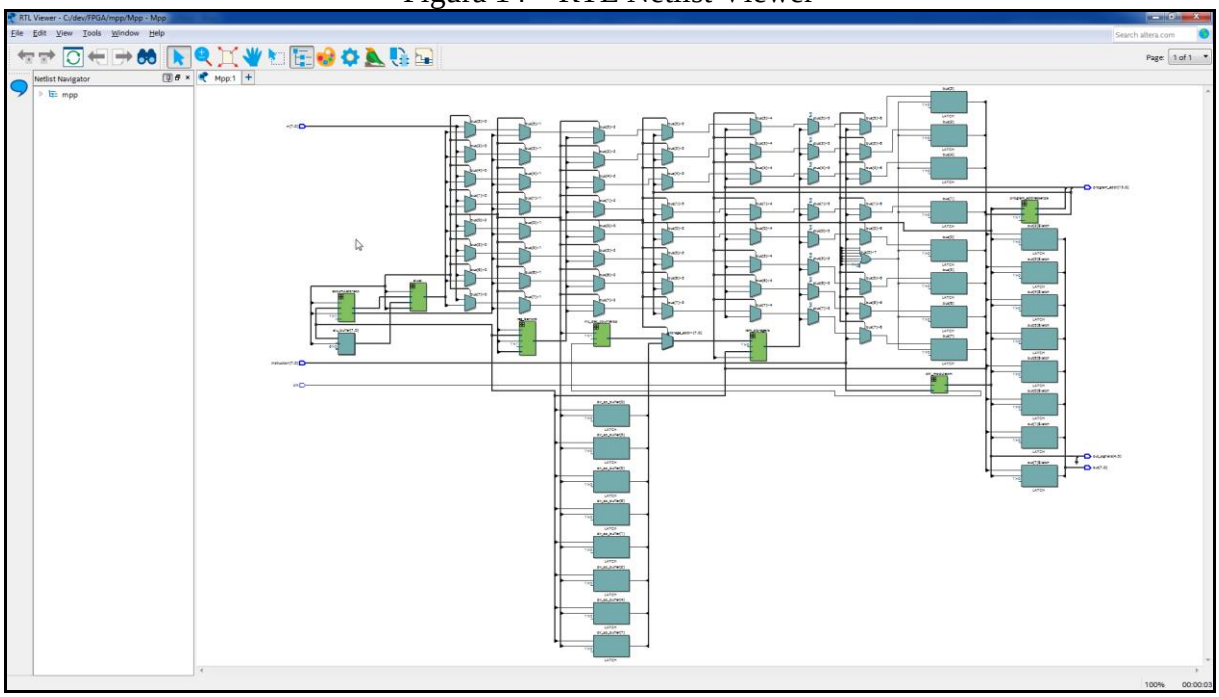


Fonte: elaborado pelo autor.

A Figura 14 mostra a tela gerada pela função RTL Netlist Viewer. Esta função mostra o circuito gerado pela compilação, e é acessível pelo menu `Tools > Netlist Viewers > RTL Viewer`. Os circuitos vistos representam o módulo `mpp`, e os símbolos verdes são os outros módulos. Nos símbolos verdes há um botão que permite expandir o símbolo e exibir os circuitos internos do módulo. É importante lembrar que o Verilog é uma linguagem para descrever hardware, e é fácil esquecer e pensar no programa como em outras linguagens. Este

esquemático tem grande utilidade para o desenvolvimento, pois é possível verificar como o código está sendo traduzido para hardware.

Figura 14 – RTL Netlist Viewer



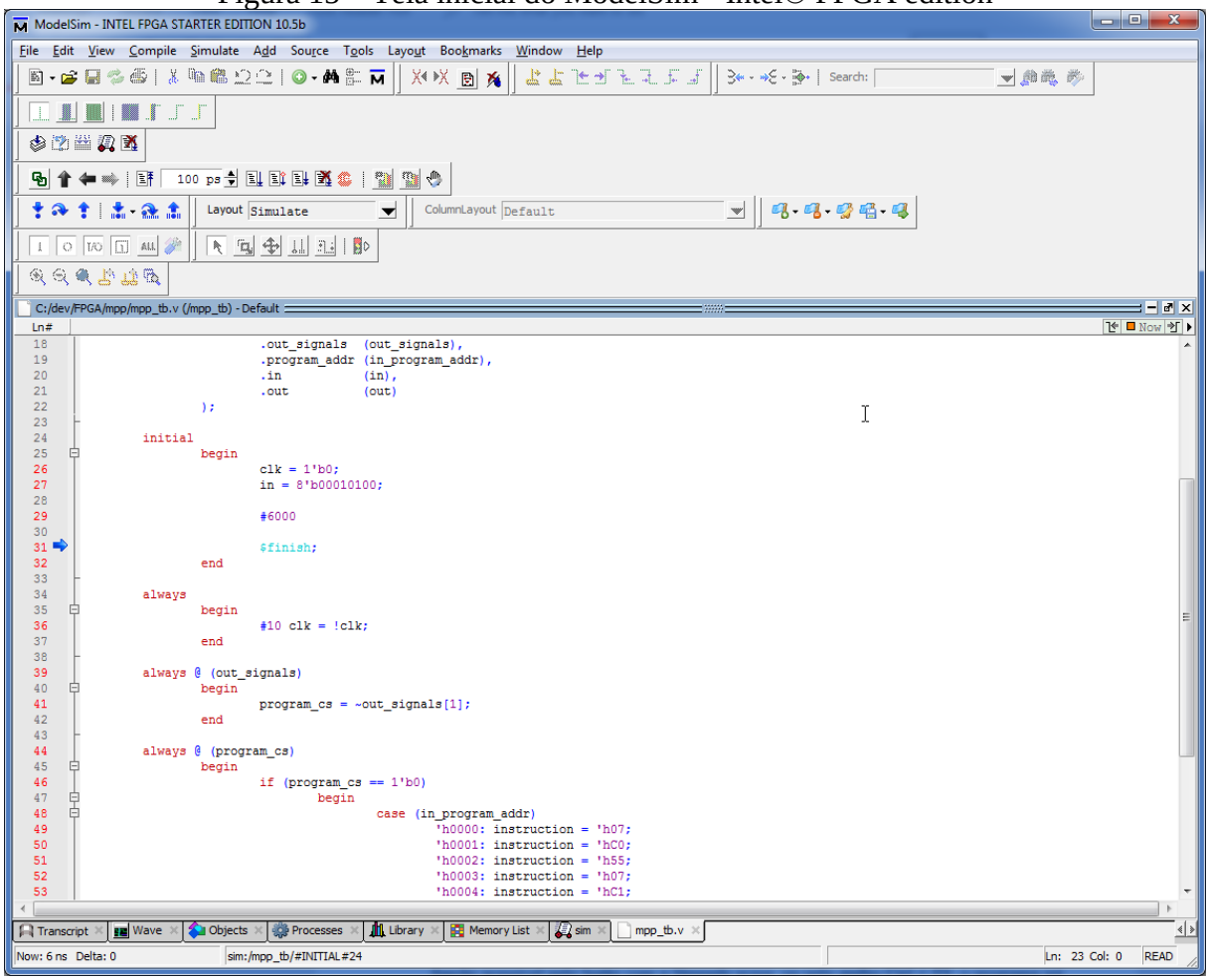
Fonte: elaborado pelo autor.

4.3.2.2 ModelSim*-Intel® FPGA edition

Programa utilizado para fazer as simulações dos modelos desenvolvidos. É uma versão especial da Intel do ModelSim* PE/DE, desenvolvido pela Mentor Graphics, que só suporta as bibliotecas de componentes da Intel e tem o tempo de execução das simulações reduzido (Intel, 2018). Utiliza um bloco de estímulo para testar o projeto. Foi utilizada a edição *Starter Edition*, pois não necessita de licença paga. As limitações em relação à versão paga são: poder simular um design com até 10 mil linhas e ter a velocidade de simulação reduzida em 30% (Intel, 2018).

O programa é dividido em várias abas, sendo que as mais utilizadas para o projeto são: *Transcript*, *Wave*, *Memory List* e *Sim*. A Figura 15 mostra a tela aberta ao iniciar o ModelSim, na aba do código do módulo de estímulo. O programa é executado pelo menu `Tools > Run Simulation Tools > RTL Simulation`, que executa o bloco de estímulo definido para o projeto. Esta tela é utilizada para fazer o *debug* do programa, e neste caso a execução está parada na instrução `$finish`, que faz com que a simulação seja pausada. É possível adicionar um *breakpoint* em algum ponto do código, reiniciar a simulação e fazer o *debug* passo a passo de alguma função.

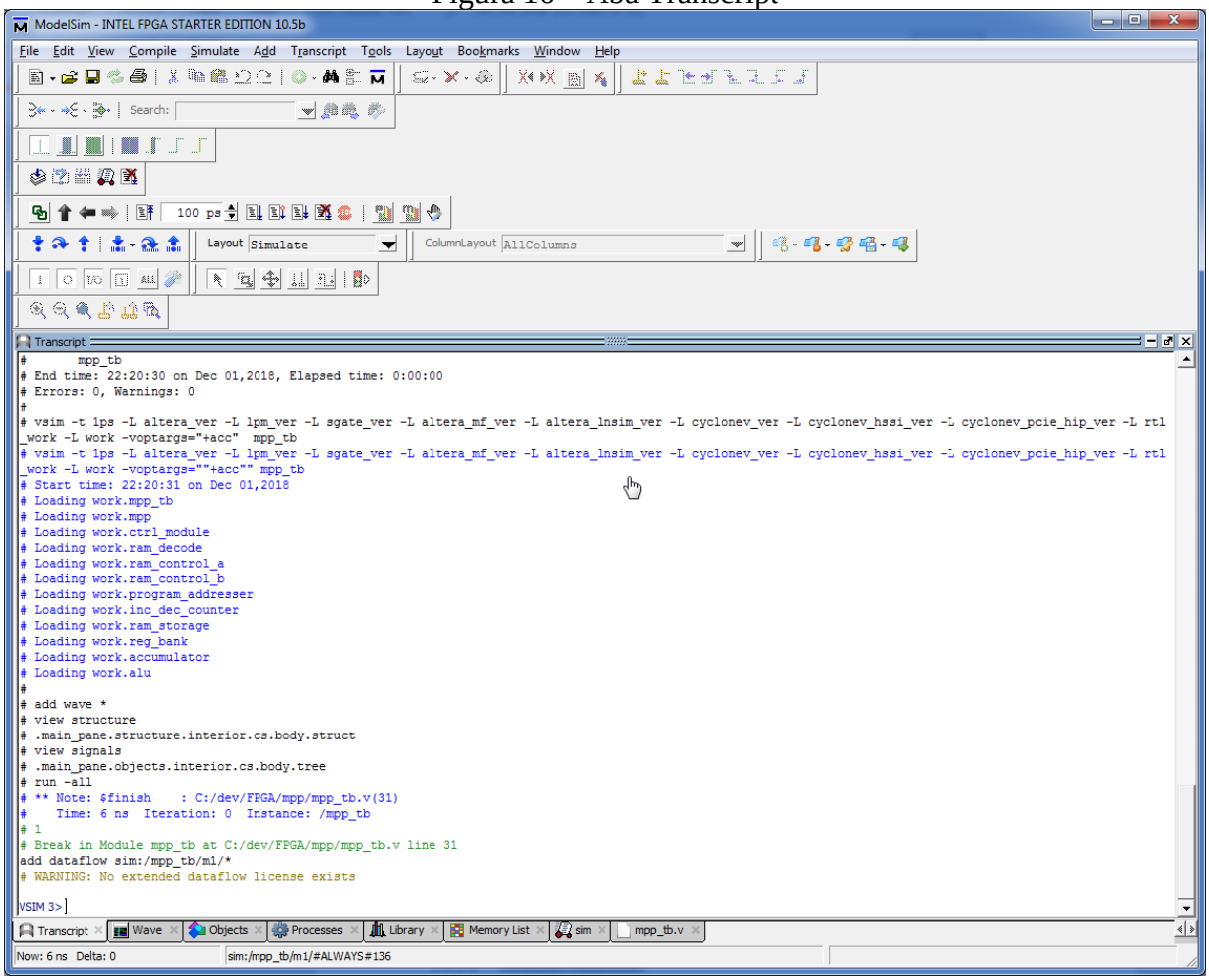
Figura 15 – Tela inicial do ModelSim*-Intel® FPGA edition



Fonte: elaborado pelo autor.

A aba *Transcript*, vista na Figura 16, é a tela que exibe o *log* da simulação. Também é possível executar comandos. Durante o desenvolvimento foram utilizados os comandos `restart -f`, que reinicia a simulação, e `run -all`, que executa a simulação. Caso haja algum erro na compilação e execução do bloco de estímulo, o erro e sua respectiva mensagem serão exibidos no *log*. Caso seja necessário fazer o *debug* de alguma função, pode-se adicionar comandos no código para imprimir mensagens no log.

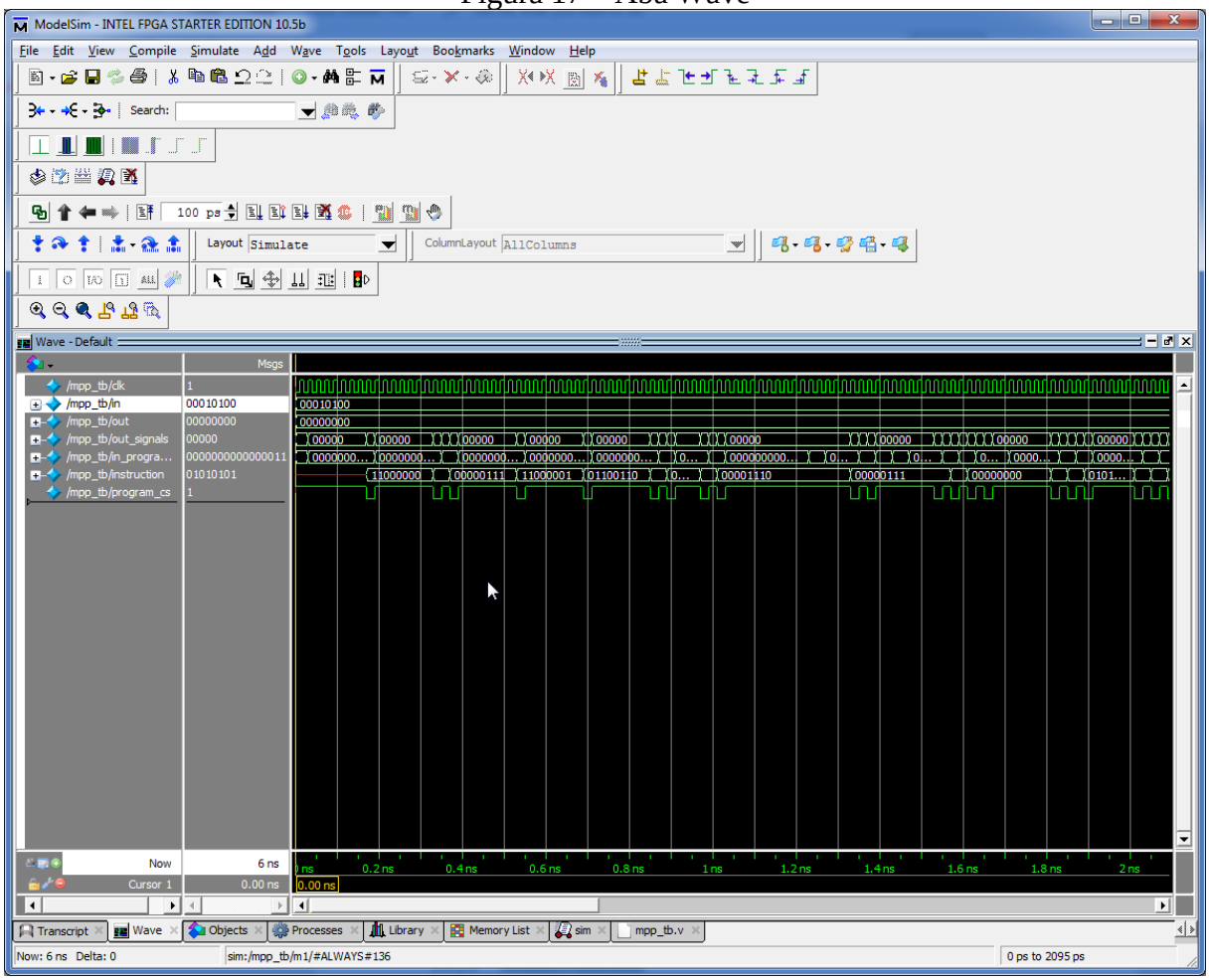
Figura 16 – Aba Transcript



Fonte: elaborado pelo autor.

A aba *Wave* é a principal para visualizar a execução do design, pois mostra as formas de onda e o estado de todas as variáveis durante toda a execução da simulação, como mostra a Figura 17. Sinais verdes significam um valor definido, vermelhos significam um valor indefinido e azuis um valor em tri-state. A simulação de um programa normalmente é bastante grande, e por isso o *scroll* horizontal é sempre utilizado. A visualização dos valores das variáveis é em binário por padrão, mas pode ser alterado para decimal e hexadecimal. A simulação sempre adiciona somente as formas de onda das variáveis do módulo de estímulo. Variáveis de outros módulos podem ser adicionadas pela aba *Sim*.

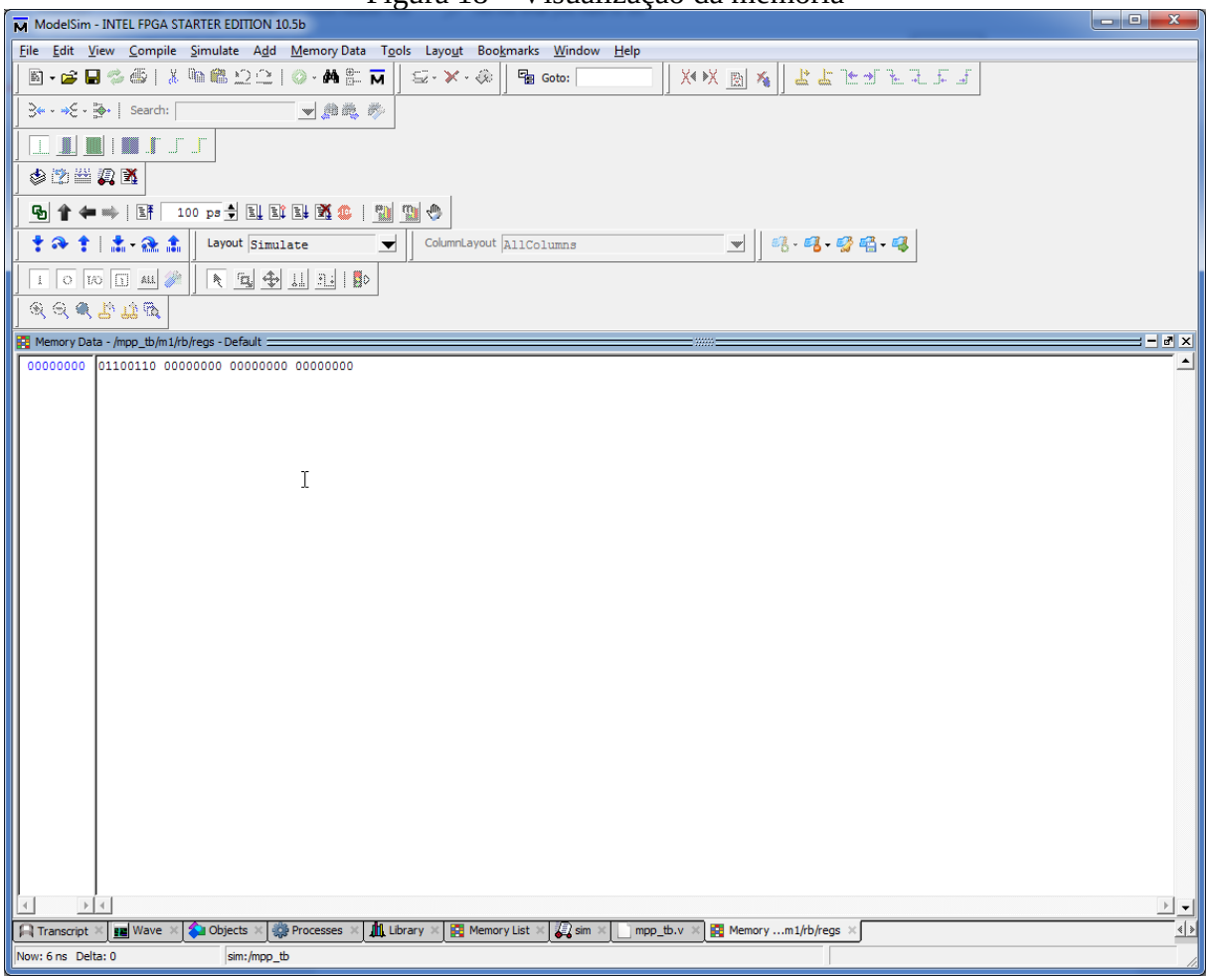
Figura 17 – Aba Wave



Fonte: elaborado pelo autor.

A aba *Memory List* é a mais simples, e serve para visualizar os conteúdos das memórias. É considerada memória as variáveis declaradas como vetor (ver seção 2.1.11). Dar um clique duplo em alguma das memórias irá abrir uma aba com todas as posições da memória e seus valores, conforme a Figura 18, que mostra os valores da memória `regs`. Os valores são exibidos por padrão em binário, mas pode ser alterado para decimal ou hexadecimal, caso necessário.

Figura 18 – Visualização da memória



Fonte: elaborado pelo autor.

A aba *Sim*, também simples, foi utilizada para adicionar as variáveis de outros módulos na aba *Wave*. Há uma lista com os módulos, organizados pela hierarquia do projeto. Para adicionar um módulo, é necessário selecionar algum dos módulos e pressionar `Ctrl + W`.

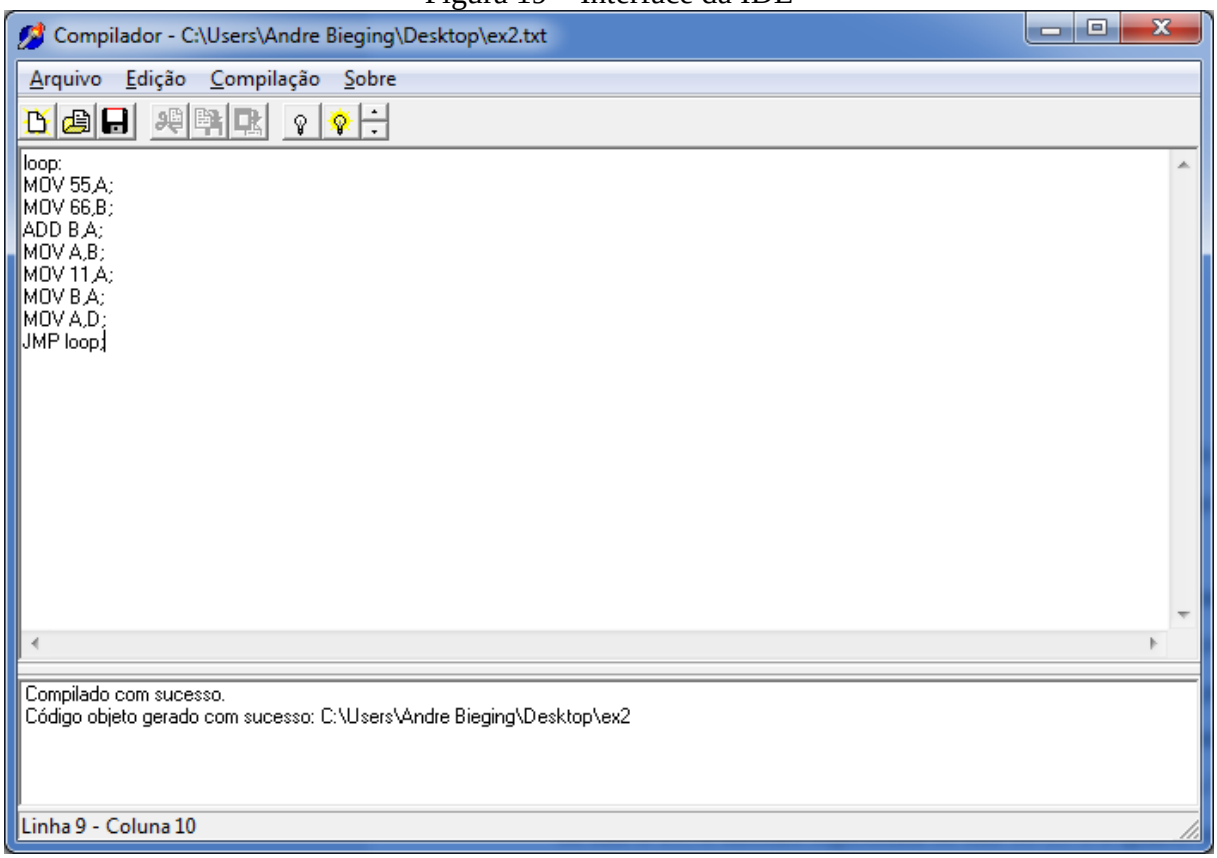
Como não são adicionadas variáveis de todos os módulos à aba *Wave*, foi sempre executado o seguinte procedimento: adicionar os módulos à aba *Wave* pela aba *Sim*, executar os comandos para reiniciar e executar a simulação. Desta forma é possível visualizar as formas de onda de todos os módulos e entender por completo o funcionamento do projeto.

4.3.2.3 MontadorMmaismais

É a IDE utilizada para desenvolver os programas para a arquitetura da M++. Foi desenvolvida por Borges (2003). Tem a funcionalidade bastante limitada, sendo que a única tela é mostrada na Figura 19, sendo possível salvar o arquivo atual, carregar um arquivo já existente, compilar o programa e gerar o código para a M++. Ao gerar o código, função acessível pelo botão com a lâmpada acesa, ou pelo atalho `Ctrl + F9`, o programa irá salvar

um arquivo do tipo `.map` com o mesmo nome do programa sendo editado. Caso o programa ainda não tenha sido salvo, uma janela de diálogo para salvar irá abrir.

Figura 19 – Interface da IDE



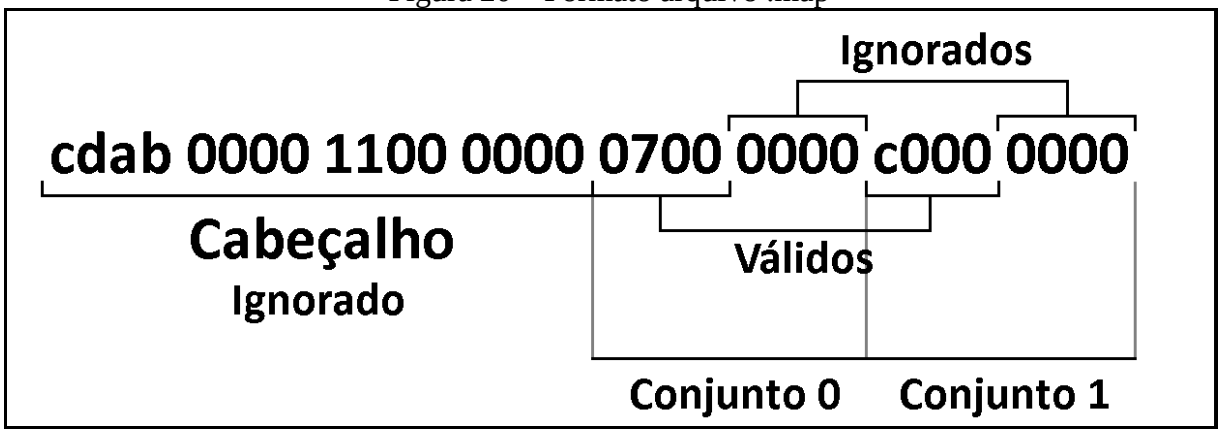
Fonte: elaborado pelo autor.

4.3.2.4 MapReader

Foi desenvolvido um programa para facilitar a geração dos programas a partir dos arquivos .map gerados pela IDE MontadorMmaismais. O programa é feito para ser executado pela linha de comando do Windows, e há vários parâmetros para gerar o arquivo de saída de forma diferente.

O arquivo .map tem o formato descrito conforme a Figura 20. São vários conjuntos de 2 bytes em hexadecimal separados por espaços em branco. Os primeiros 8 bytes são o cabeçalho do arquivo, e o restante são os bytes do programa. Cada instrução gerada é composta por 2 conjuntos de 2 bytes, ou seja, tem um total de 32 bits, mas como são necessários apenas 8 bits para o programa, o segundo conjunto é sempre ignorado.

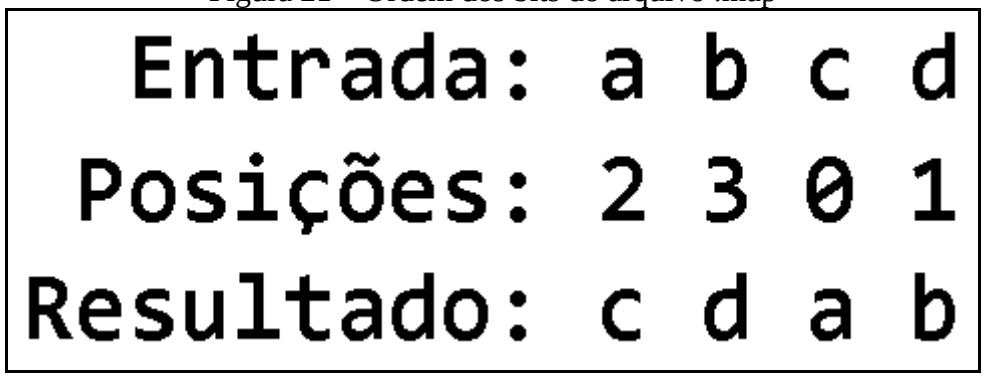
Figura 20 – Formato arquivo .map



Fonte: elaborado pelo autor.

Os bytes da instrução têm a ordem invertida, sendo que é necessário fazer a troca para obter o valor correto. A Figura 21 mostra como ocorre a correção das posições. A palavra restante do conjunto, após ignorar os dois bytes, tem quatro caracteres e cada caractere equivale a 4 *bits*. O primeiro caractere é armazenado nos *bits* 4-7, o segundo caractere é armazenado nos *bits* 0-3, o terceiro é armazenado nos *bits* 12-15 e o quarto nos *bits* 8-11. Como exemplo, caso a palavra lida seja `abcd`, o valor resultante da leitura será `cdab`.

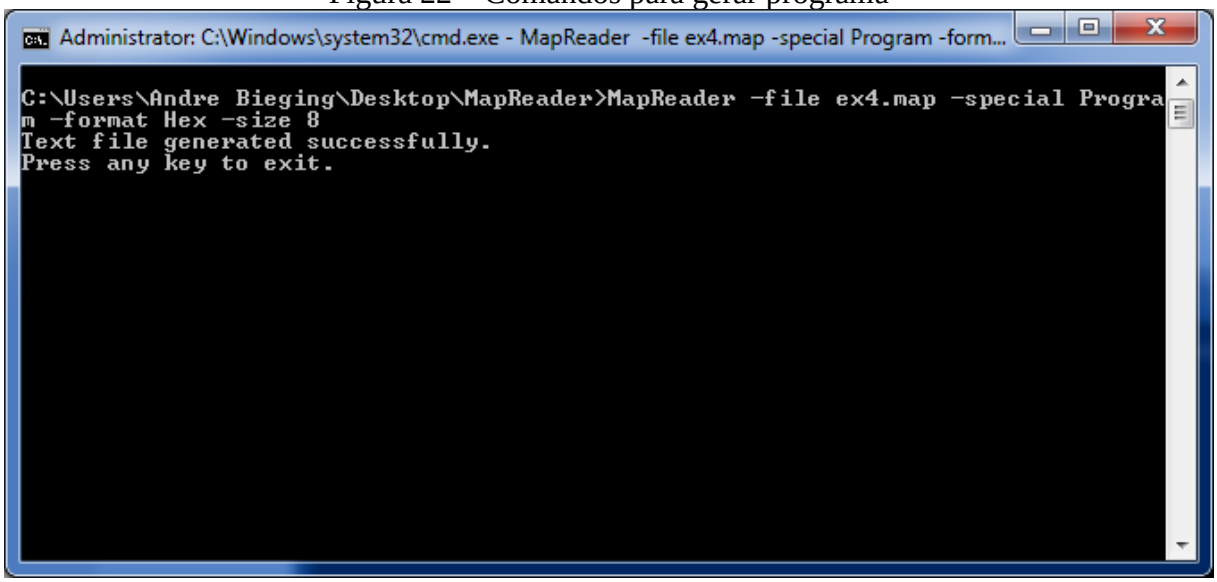
Figura 21 – Ordem dos bits do arquivo .map



Fonte: elaborado pelo autor.

A Figura 22 mostra os argumentos necessários para gerar o programa já em formato Verilog. O argumento `-file` serve para especificar qual arquivo será processador. O argumento `-special` é utilizado para especificar qual o formato de saída do arquivo gerado. Neste caso, com o valor igual a `Program`, o programa irá gerar o programa pronto em Verilog. O argumento `-format` define se os valores de saída serão salvos em hexadecimal ou em binário, e o argumento `-size` define com quantos *bits* serão salvos os valores do arquivo gerado. O programa irá imprimir uma mensagem caso o arquivo seja gerado com sucesso e irá salvá-lo no mesmo diretório com a extensão `txt`.

Figura 22 – Comandos para gerar programa



Fonte: elaborado pelo autor.

4.3.3 Operacionalidade da implementação

Para testar o projeto, foi utilizado o programa descrito pelo Quadro 32, que possui as principais funções a serem validadas, como operações de movimentação de dados, operações aritméticas, exibição de dados na saída do usuário e instruções de `JMP` e `CALL/RET`.

Quadro 32 – Programa de teste

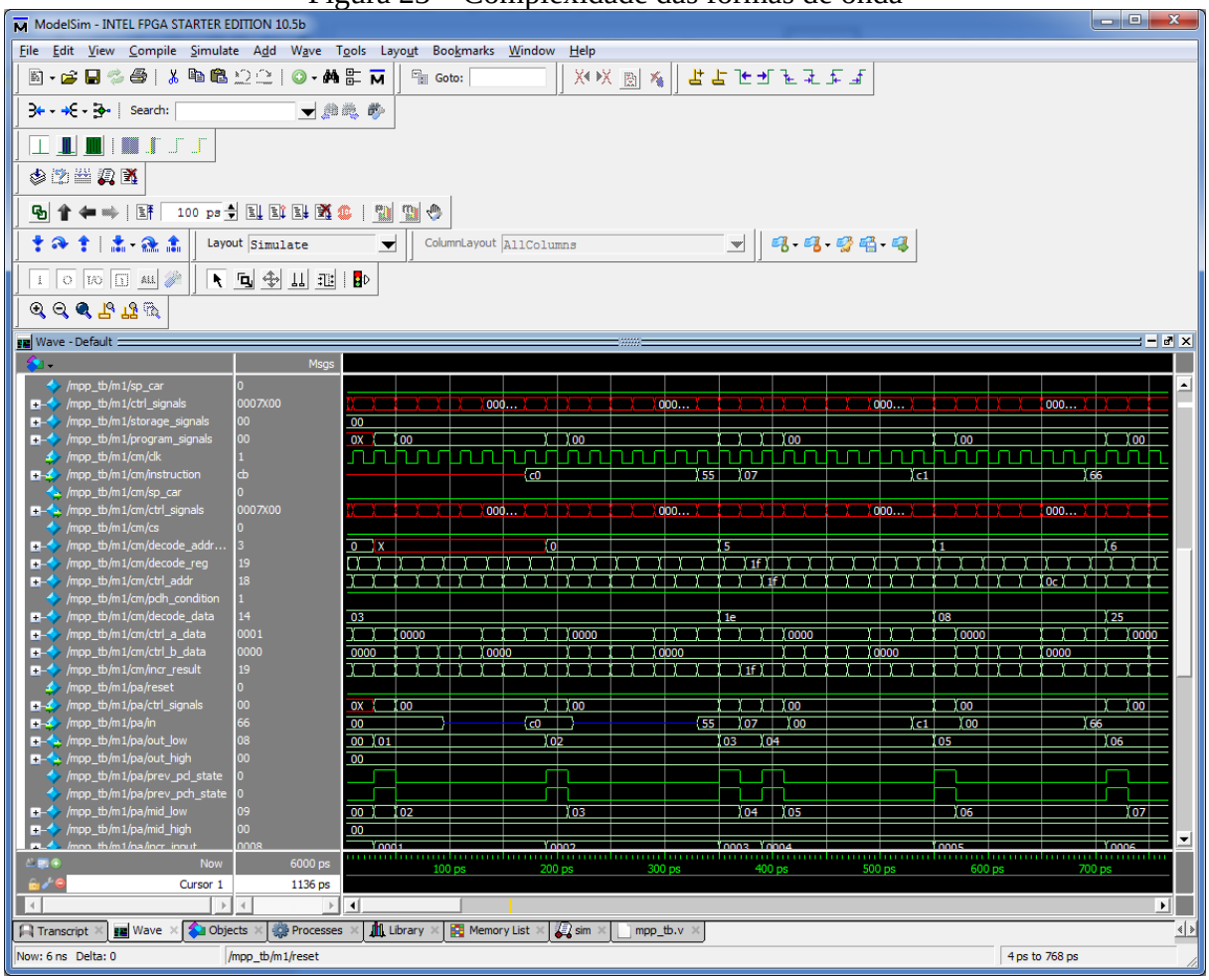
```
loop:
MOV 55,A;
MOV 66,B;
ADD B,A;
MOV A,B;
MOV A,OUT1;
CALL MOVE;
JMP LOOP;

MOVE:
RET;
```

Fonte: elaborado pelo autor.

A operação do usuário no momento é feita pelo simulador, e considerando a quantidade de variáveis em todo projeto e que todas devem ser adicionadas à aba Wave para melhor observação da execução, a usabilidade se torna bastante complexa. A Figura 23 mostra uma parte das formas de onda exibidas. Por causa dessa complexidade, serão mostrados apenas trechos importantes da simulação e apenas os sinais relacionados à determinada operação.

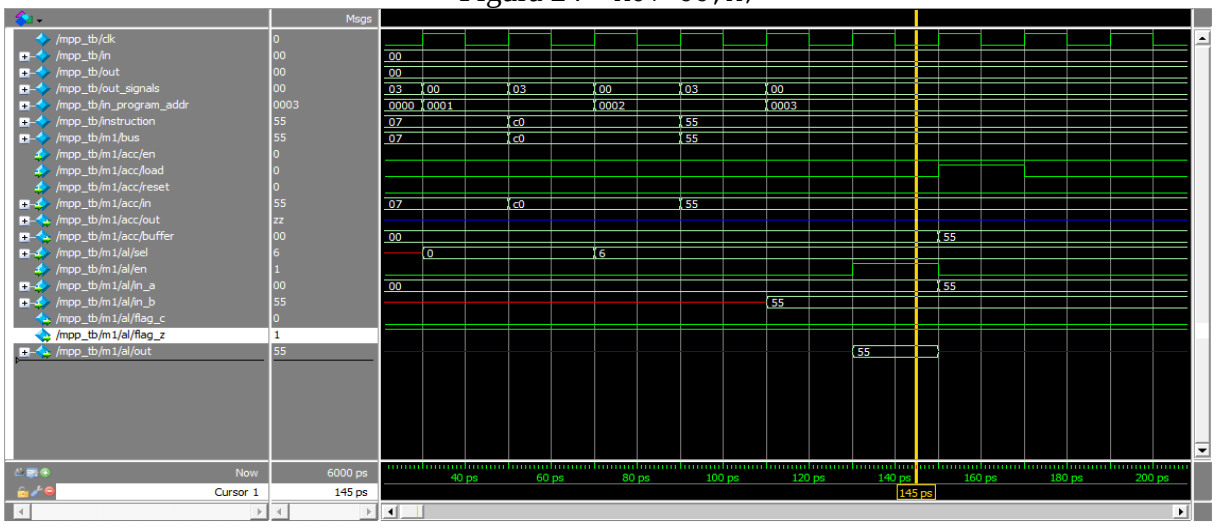
Figura 23 – Complexidade das formas de onda



Fonte: elaborado pelo autor.

A primeira linha do código é a movimentação do valor `0x55` para o acumulador. Pode ser visto na Figura 24 que no ponto indicado pela linha vertical, a entrada do acumulador é igual ao barramento, e o sinal de carga do acumulador é acionado, salvando o valor da entrada.

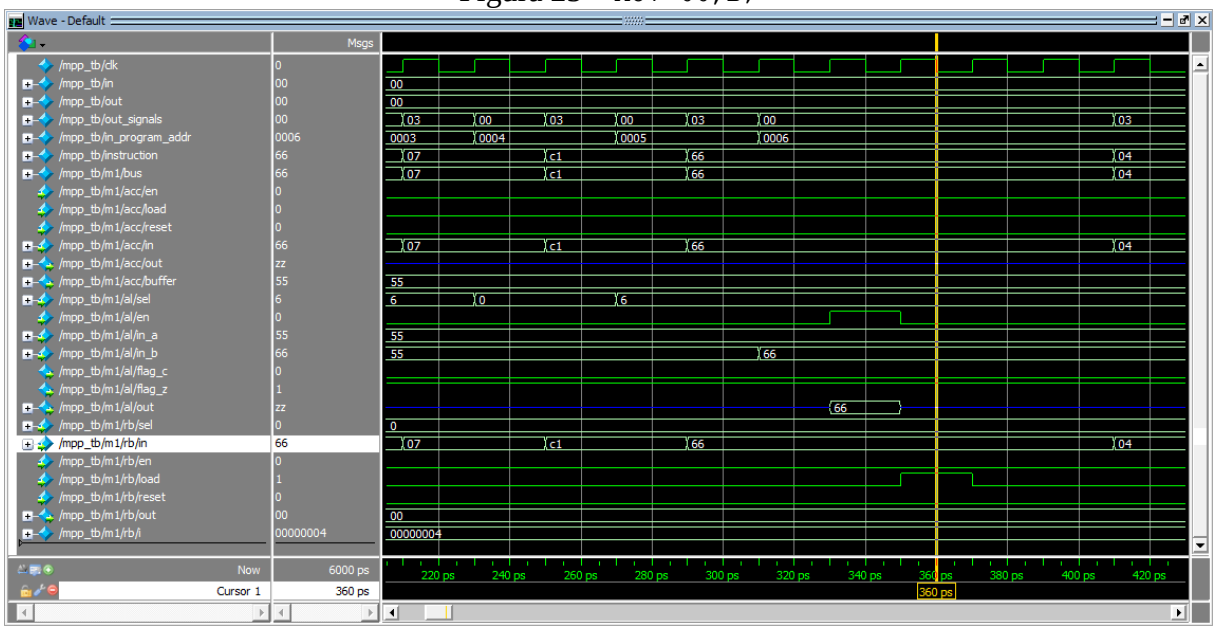
Figura 24 – `MOV 55,A;`



Fonte: elaborado pelo autor.

A segunda linha movimenta `0x66` para o registrador B. Na Figura 25, no ponto indicado pela linha vertical, a entrada do banco de registradores recebe o valor do barramento, `0x66`, e o sinal de carga é acionado, salvando o valor da entrada no registrador B.

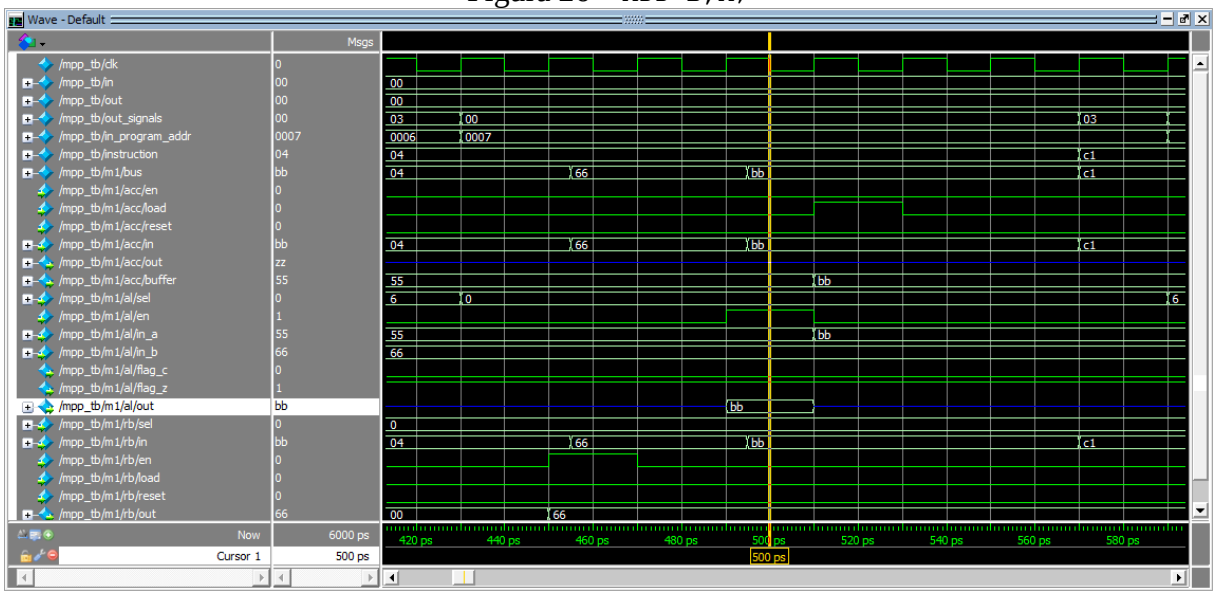
Figura 25 – `MOV 66,B;`



Fonte: elaborado pelo autor.

Na Figura 26 é executada a soma entre o valor do acumulador e o valor do registrador B. No ponto indicado pela linha vertical, é possível ver que as duas entradas da ULA estão com os valores corretos. Assim, quando é acionado o sinal de `enable`, o valor resultante, `0xBB`, é jogado no barramento.

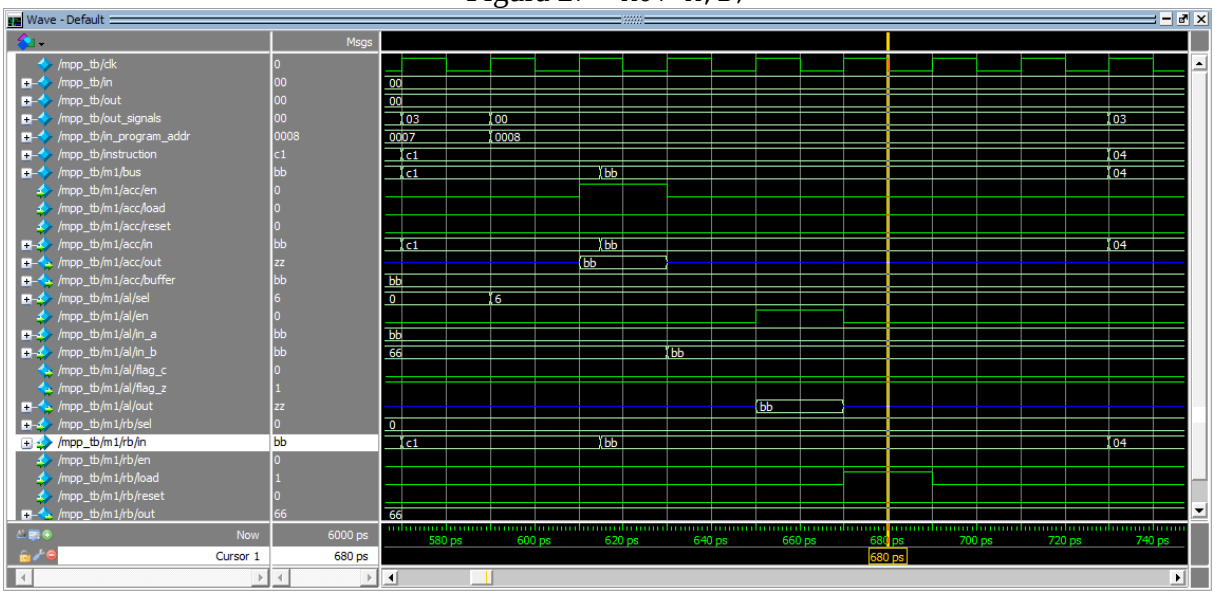
Figura 26 – `ADD B,A;`



Fonte: elaborado pelo autor.

Na Figura 27 há a operação de movimentação entre dois registradores. O barramento recebe o valor armazenado no acumulador, `0xBB`, e no ponto indicado pela linha vertical, o banco de registradores recebe o sinal de carga para salvar o valor de entrada.

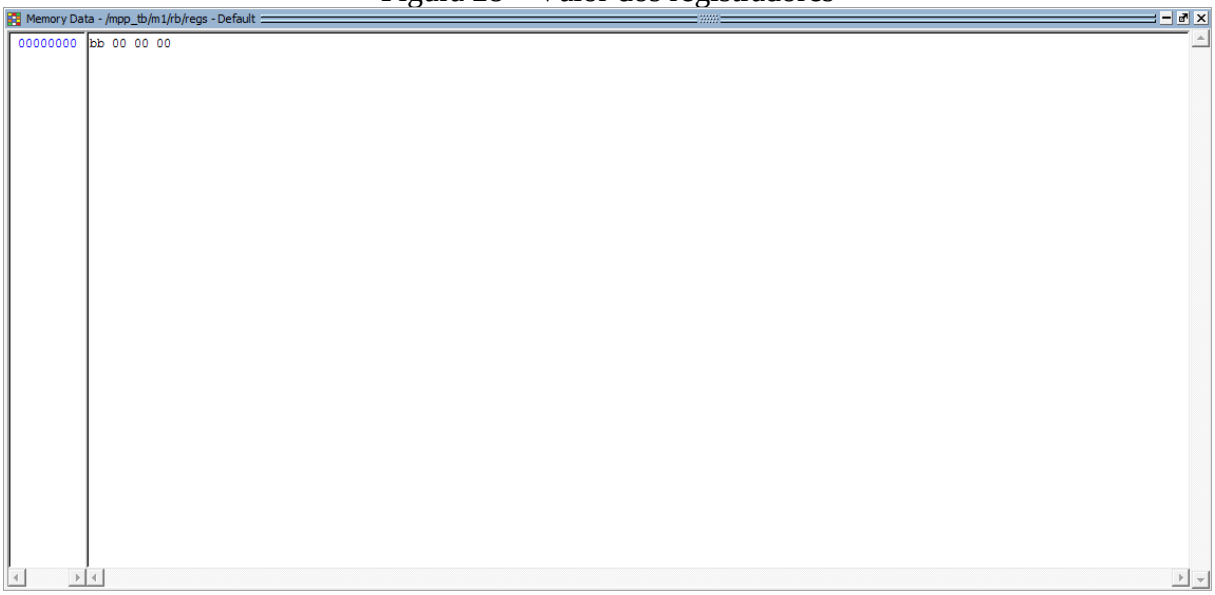
Figura 27 – `MOV A,B;`



Fonte: elaborado pelo autor.

A Figura 28 mostra os valores atualizados dos registradores após a execução da última instrução.

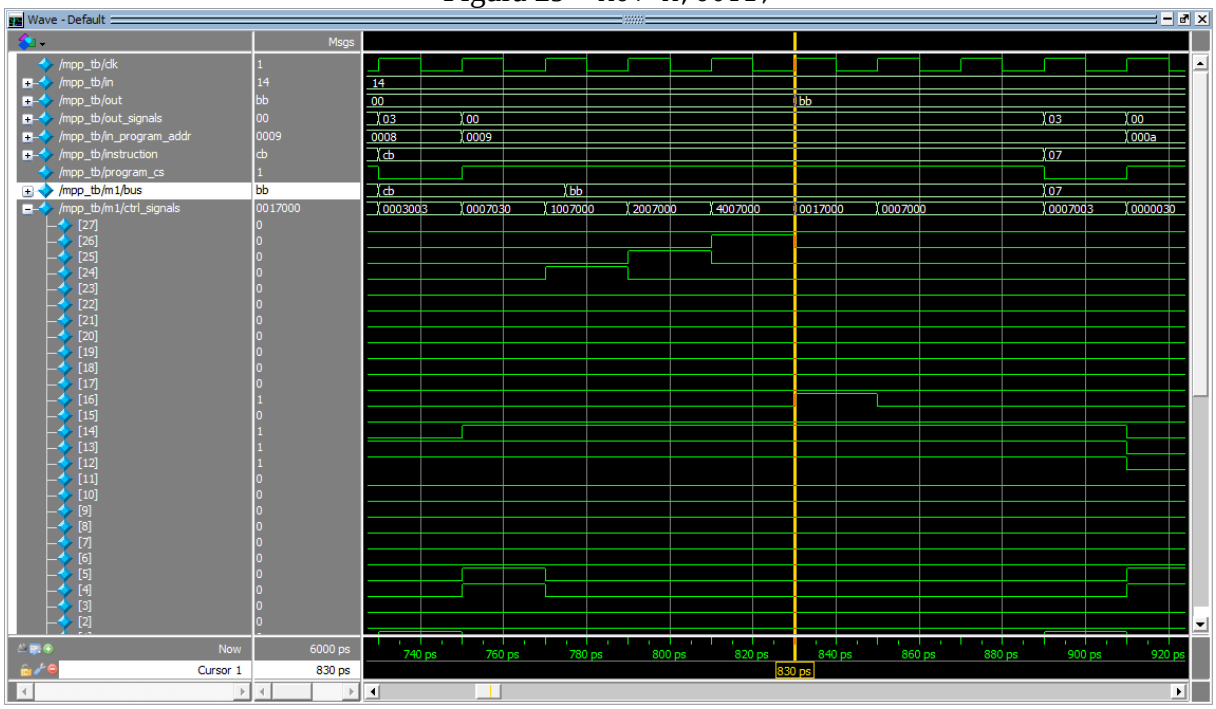
Figura 28 – Valor dos registradores



Fonte: elaborado pelo autor.

A instrução de jogar o valor do acumulador na saída é mostrada na Figura 29. Durante a execução, é acionado o sinal `ctrl_signals[16]`, que joga o valor atual do barramento na saída. No ponto indicado pela linha vertical, é possível ver que a variável `out` já recebeu o valor `0xBB`.

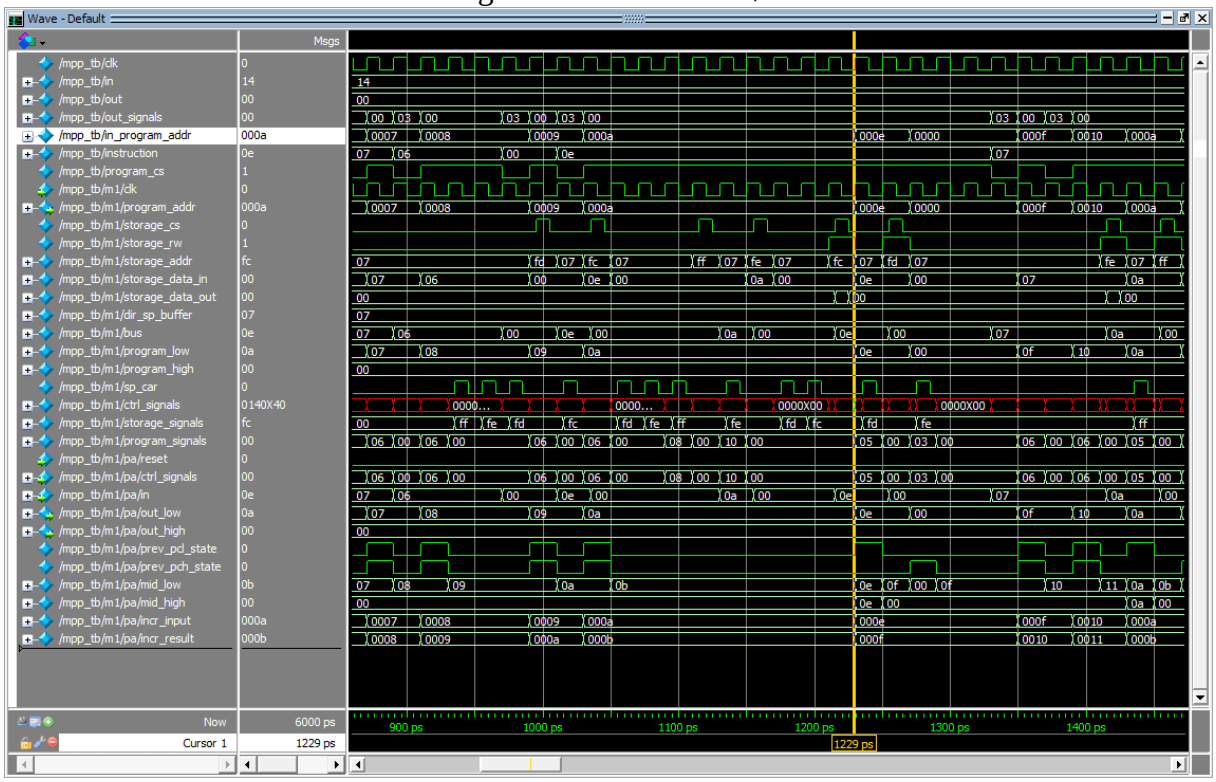
Figura 29 – `MOV A,OUT1;`



Fonte: elaborado pelo autor.

Na Figura 30 é executada a instrução `CALL`. Pode ser visto que no ponto indicado pela linha vertical a variável `in_program_addr`, que controla o endereço atual do programa, recebe o novo valor, `0x0E`, que é a posição inicial da sub-rotina.

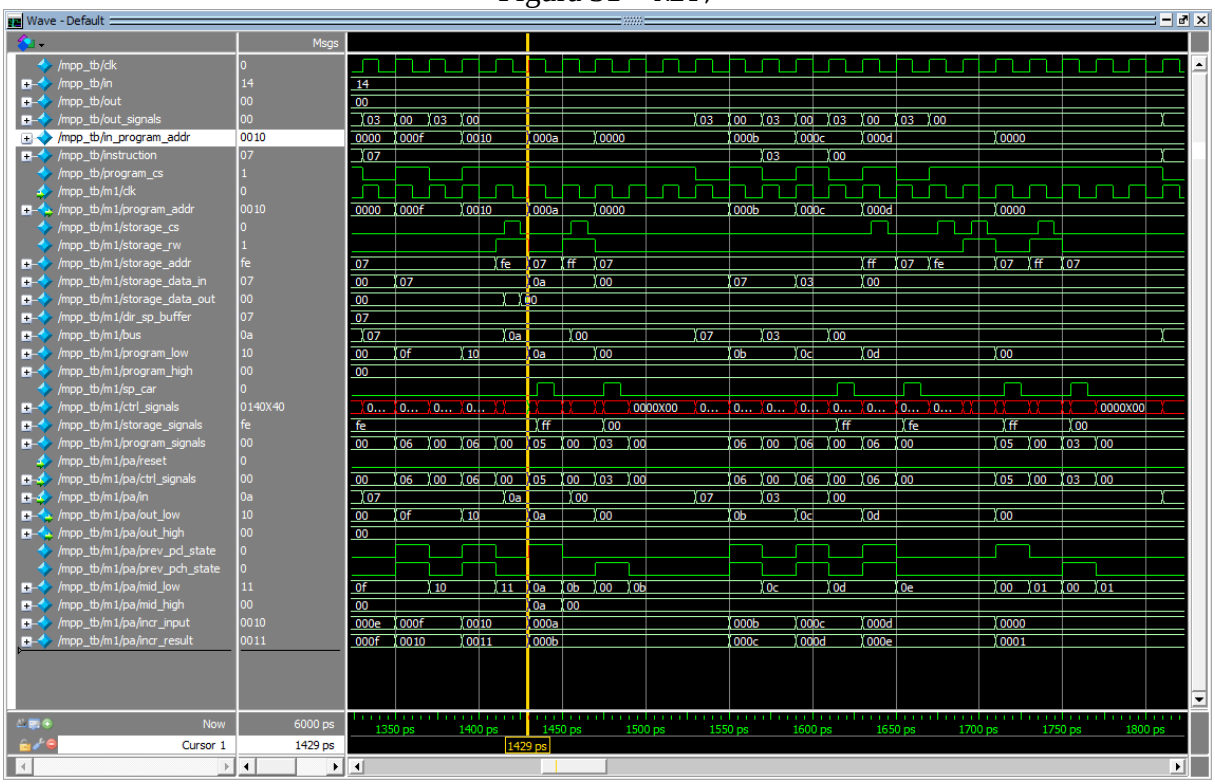
Figura 30 – `CALL MOVE;`



Fonte: elaborado pelo autor.

O comando RET é executado conforme visto na Figura 31. Após executar a instrução do endereço de programa `0x10`, o programa retorna ao endereço `0x0A`, no qual estava antes de executar o comando `CALL`.

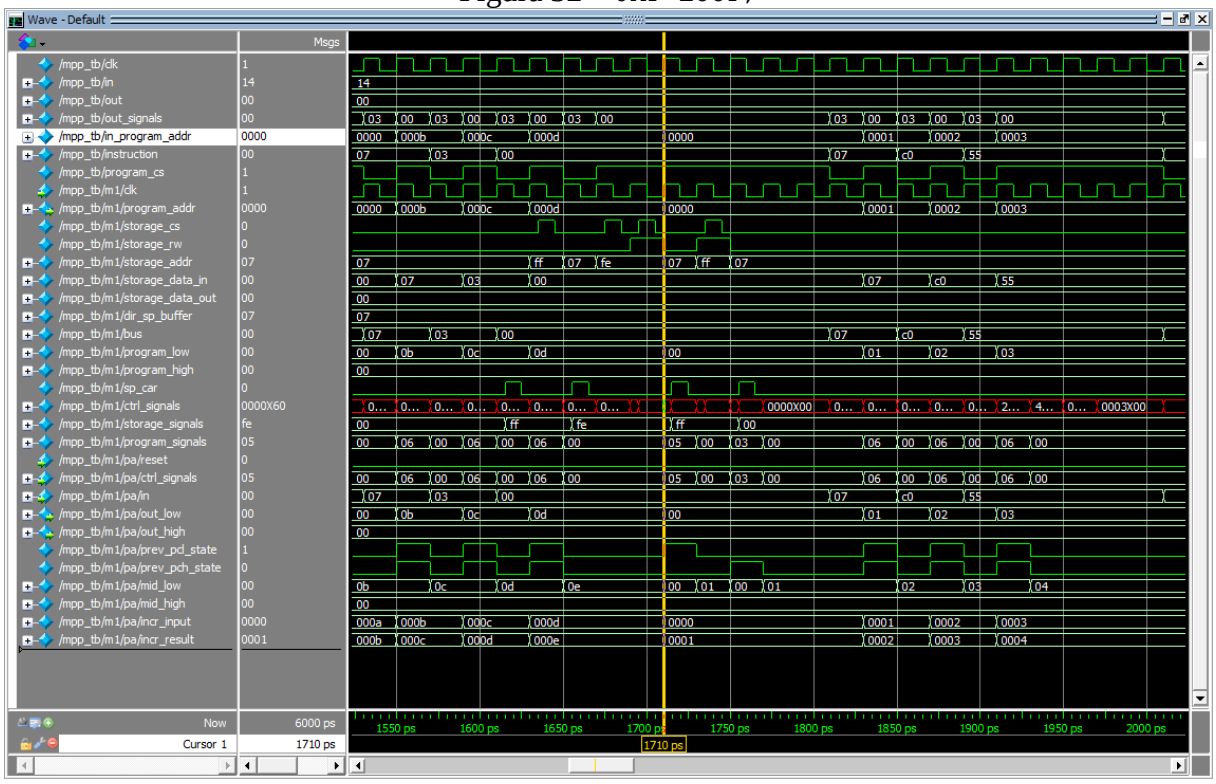
Figura 31 – `RET;`



Fonte: elaborado pelo autor.

Na Figura 32 ocorre o `JMP` para o início do programa. Pode ser visto pelo ponto indicado pela linha vertical que o endereço do programa voltou a ser `0x00`. Graças a esta instrução, o programa é executado repetidamente, até que o módulo de estímulo chame o comando $`finish`, após 6000 unidades de tempo de execução.

Figura 32 – `JMP LOOP;`

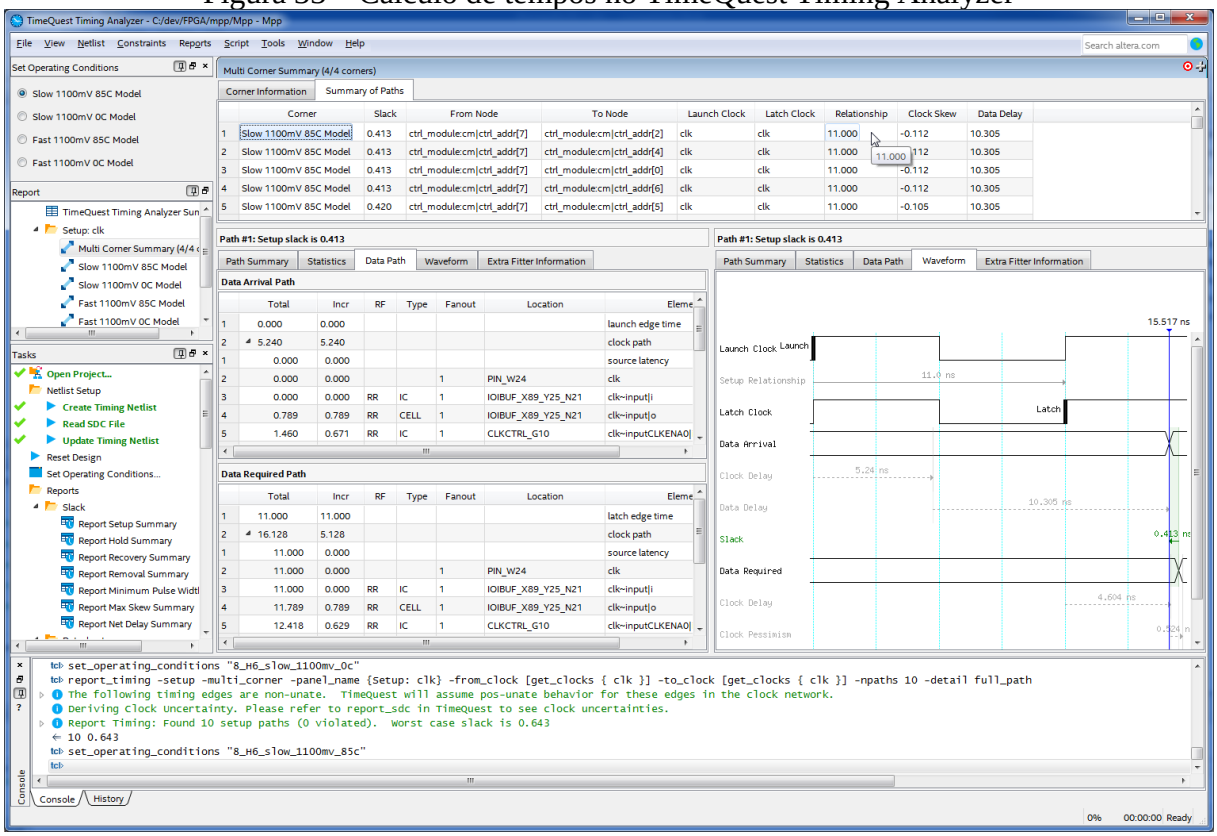


Fonte: elaborado pelo autor.

4.4 ANÁLISE DOS RESULTADOS

Pelo programa testado, é possível ver que a execução das instruções definidas funciona corretamente. Conforma a Figura 33, foi executado um cálculo de tempos pelo TimeQuest Timing Analyzer, uma extensão do Quartus, onde foi calculado que o clock mínimo para que o projeto funcione sem erros de *Race Condition* é 11ns, o que equivale a uma frequência máxima de operação de aproximadamente 90MHz.

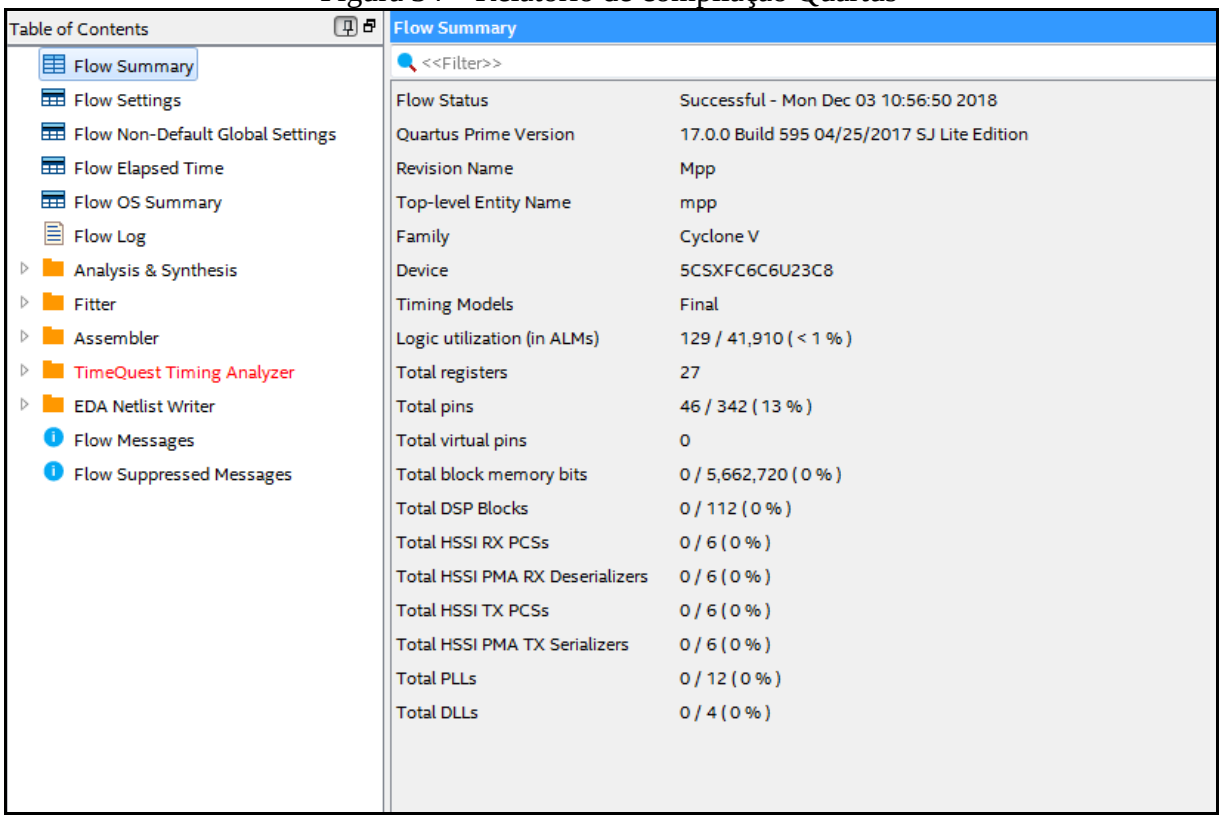
Figura 33 – Cálculo de tempos no TimeQuest Timing Analyzer



Fonte: elaborado pelo autor.

Na questão de elementos lógicos do FPGA, o projeto utilizou apenas 129 dos mais de 41000 disponíveis, conforme o relatório de compilação do Quartus exibido na Figura 34, que representa menos de 1% do total disponível.

Figura 34 – Relatório de compilação Quartus



Fonte: elaborado pelo autor.

No Quadro 33 é apresentado, de forma comparativa, diversas características dos trabalhos correlatos e do projeto desenvolvido, onde as linhas representam as funcionalidades e as colunas os trabalhos.

Quadro 33 – Comparação dos trabalhos correlatos

correlatos / características	AYEH et al., (2008)	PABLO et al., (2016)	ZALAVA et al., (2015)	M++ FPGA
fabricante	Xilinx	Xilinx	Xilinx	Altera
tecnologia	FPGA	FPGA	FPGA	FPGA
número de instruções	4	29	29	14
arquitetura	Própria	Harvard	Harvard	Harvard
set de instruções	Próprio	RISC	RISC	Próprio
frequência	~95MHz	~40MHz	Não inf.	90MHz

Fonte: elaborado pelo autor.

Enquanto todos os trabalhos correlatos utilizaram ferramentas do fabricante Xilinx, o projeto foi desenvolvido com ferramentas da Altera. O motivo da escolha do fabricante foi a familiaridade do autor com suas respectivas ferramentas, e pela ampla quantidade de documentação introdutória de qualidade.

O primeiro trabalho possui um set de instruções muito inferior aos outros dois trabalhos e é em si um processador mais simples capaz de executar apenas operações de soma de subtração. Os outros dois possuem diversas outras instruções e operações, totalizando 29

ao todo, enquanto a M++ possui 14 instruções. Mesmo não tendo tantas funcionalidades, a arquitetura da M++ é bastante capaz.

Assim como o segundo e o terceiro trabalhos, a M++ também utiliza a arquitetura de alto nível Harvard, enquanto o primeiro trabalho usa uma arquitetura própria, mais simples. A M++ e o primeiro trabalho tem um set de instruções próprio, enquanto os outros dois trabalhos tem o set de instruções RISC. Dois dos trabalhos conseguiram fazer seus microprocessadores funcionarem em frequências de dezenas de MHz. Com o simulador da M++ atual, é possível alcançar apenas 50Hz, mas com o trabalho desenvolvido em FPGA, foi alcançado uma frequência de até 90MHz, um avanço extremamente grande.

5 CONCLUSÕES

Conclui-se que é possível implementar a arquitetura da M++ em Verilog e conseguir um design com uma performance considerável. O trabalho atingiu os objetivos de carregar um programa da memória, processar as entradas e produzir saídas conforme o esperado. Também foi criada uma documentação detalhada da arquitetura, para que possa ser utilizada como material de aula. Foi verificado que é possível incrementar a arquitetura com as instruções extras da M+++, e que não há grande complexidade em adicionar novas instruções. As ferramentas da Altera, mesmo que sendo nas versões não pagas, atenderam muito bem ao desenvolvimento, fornecendo todas as funções necessárias. O desenvolvimento deste trabalho é importante pois mostra um *approach* simples à área de arquitetura de computadores e de engenharia da computação, e pode abrir portas para que seja lecionado sobre este assunto no curso. Por fim, será uma prova de que a arquitetura da M++, criada na FURB e desenvolvida para funcionar apenas dentro de um simulador, não só funciona em um chip real, como também tem uma performance na casa dos MHz. É importante notar que com o desenvolvimento em Verilog, a arquitetura pode ser facilmente melhorada e expandida. No momento, o desenvolvimento ainda tem algumas deficiências no código que poderiam ser melhoradas.

5.1 EXTENSÕES

Esta seção mostra algumas das melhorias que poderiam ser aplicadas ao projeto. São elas:

- adicionar instruções da M+++;
- estudar a arquitetura RISC e verificar se a arquitetura da M++ pode ser melhorada em algum ponto;
- barramento de 16 *bits*: permitiria um número muito maior de instruções, operações da ULA e registradores;
- hierarquia: A hierarquia de módulos poderia ser melhorada, evitando ter a lógica final de funcionamento em módulo de alto nível;
- uso de softwares *open-source*: caso este projeto seja aplicado para lecionar sobre o assunto, poderia ser levantada a viabilidade de usar softwares *open-source*;

- estruturar documentação: a documentação desenvolvida explica em bastante detalhe a arquitetura, mas poderia ser melhorada e estruturada para ser usada em sala de aula.

REFERÊNCIAS

AGARWAL, T. **Basic FPGA Architecture and its Applications**. [S.l.], [2017]. Disponível em: <https://www.edgefx.in/fpga-architecture-applications/>. Acesso em: 8 jun. 2018.

AYEH, Eric et al. **FPGA Implementation of an 8-bit Simple Processor**. Denton, [2008]. Disponível em: <https://www.researchgate.net/publication/4342279>. Acesso em: 4 abril 2018.

BORGES, Jonathan M.. **CONSTRUÇÃO DE UMA UCP HIPOTÉTICA M++**. Blumenau, [2003]. Disponível em: <http://www.inf.furb.br/~maw/mmaismais/artigos/artigo.pdf>. Acesso em: 6 abril 2018.

BORGES, Jonathan M.. **Montador**. Blumenau, [2003]. Disponível em: <http://www.inf.furb.br/~maw/mmaismais/montador.shtml>. Acesso em: 11 nov. 2018.

CUTRESS, I. **NVIDIA's DGX-2: Sixteen Tesla V100s, 30 TB of NVMe, only $400K**. [S.l.], [2018]. Disponível em: < https://www.anandtech.com/show/12587/nvidias-dgx2-sixteen-v100-gpus-30-tb-of-nvme-only-400k>. Acesso em: 7 jun. 2018.

PABLO, S. de. et al. **A very simple 8-bit RISC processor for FPGA**. Valladolid, [2016]. Disponível em: <https://www.researchgate.net/publication/267766934>. Acesso em: 4 abril 2018.

IEEE. **IEEE 1364-2001**. [2001]. Disponível em: < https://standards.ieee.org/standard/1364-2001.html>. Acesso em: 2 dez. 2018.

INTEL. **ModelSim*-Intel® FPGA Edition Software**. [S.l.], [2008]. Disponível em: < https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html>. Acesso em: 2 dez. 2018.

PALNITKAR, Samir. **Verilog HDL**: A Guide to Digital Design and Synthesis, 2 ed. Prentice Hall PTR, 2003

Vaughan-Nichols, S. **A super-fast history of supercomputers: From the CDC 6600 to the Sunway TaihuLight**. [S.l.], [2017]. Disponível em: https://www.hpe.com/us/en/insights/articles/a-super-fast-history-of-supercomputers-from-the-cdc-6600-to-the-sunway-taihulight-1711.html#>. Acesso em: 7 jun. 2018.

Zavala, Antonio H. et al. **Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning**. Querétaro, [2015]. Disponível em: <http://www.scielo.org.mx/pdf/cys/v19n2/v19n2a13.pdf>. Acesso em: 4 abril 2018.