UNIVERSIDADE FEDERAL DE SANTA CATARINA

Guilherme Frick de Oliveira

Projeto e implementação em VHDL para FPGAs de uma arquitetura baseada em RISC-V de ciclo único

Florianópolis

2019

Guilherme Frick de Oliveira

Projeto e implementação em VHDL para FPGAs de uma arquitetura baseada em RISC-V de ciclo único.

Projeto final da disciplina Sistemas Digitais e Dispositivos Lógicos Reconfiguráveis, do Programa de Pós Graduação em Engenharia Elétrica, Mestrado em Sistemas Embarcados.

Prof. PhD. Eduardo Augusto Bezerra

Florianópolis

2019

RESUMO

O projeto da arquitetura de processador RISC-V tem como ponto de partida as especificações como o tamanho e o tipo de dados que irão ser processador. A arquitetura do conjunto de instruções (ISA) alvo neste projeto é o RV32I (registros de inteiros de base 32bits).Será implementada em VHDL (Vhsic Hardware Description Language) para FPGAs o modelo básico do de processador RISC-V de ciclo único. Estudou-se as instruções a serem implementadas, identificando as unidades necessárias para o desenvolvimento de um processador. Projetou-se os componentes necessários para a execução prática do núcleo, sendo eles a memória de programa e dados, banco de registradores, unidade lógica aritmética, o sistema de controle e um multiplicador paralelo. A validação é dada através de simulações, e pela execução de um determinado conjunto de instruções e alguns programas de aplicação escritos em Assembly.

Palavras-chave: RISC-V, RV64I, ISA, VHDL,FPGA, Ciclo único.

ABSTRACT

The design of the RISC-V processor architecture takes as its starting point the specifications as the size and type of data that will be processor. The architecture of the target instruction set (ISA) in this project is RV32I (32bits base integer registers). The Vhsic Hardware Description Language (VHDL) for FPGAs is implemented in the basic single-cycle RISC-V processor model. We studied the instructions to be implemented, identifying the necessary units for the development of a processor. The components necessary for the practical execution of the core were designed, being program and data memory, register bank, arithmetic logic unit, control system and a parallel multiplier. Validation is given through simulations, and through the execution of a set of instructions and some application programs written in Assembly.

Keywords: RISC-V, RV32I, ISA, VHDL,FPGA, Single Cycle.

SUMÁRIO

1 INTRODUÇÃO

A arquitetura do conjunto de instruções (Intstructions Set Architecture - ISA) assim como a interface hardware-software, constituem a interface principal de um sistema computacional. Entretanto, nos sistemas computacionais modernos as ISAs popularmente conhecidas são exclusivas de seus proprietários. Um padrão de ISA livre e aberto, permite que uma nova era de inovação de processadores de propósito geral seja alcançada por meio de colaboração aberta, modo que seja oferecido um suporte mais fácil de uma ampla variedade de sistemas operacionais, fornecedores de software e desenvolvedores de ferramenta. Com esse propósito, a fonte aberta de hardware padrão RISC-V não depende de um único fornecedor, portanto, suporta um potencial ilimitado para crescimento futuro. [1]

O RISC-V cria e aprimora as arquiteturas originais do RISC (Reduced Instruction Set Computer). O resultado é uma ISA limpa e simples e que é bem adequada para baixo consumo de energia em sistema embarcados. [2] . RISC-V é projetado como uma arquitetura modular, com variantes cobrindo espaços de endereços de 32 bits, 64 bits e 128 bits e também oferece opções de extensões de conjuntos de instruções como ponto-flutuante, multiplicadores, etc (RISC-V Standard Extensions). O conjunto de instruções de base inteira é enxuto o suficiente exigindo menos de 50 níveis de instruções de hardware para ser adequado para fins educacionais e para muitos processadores incorporados incluindo as unidades de controle de aceleradores personalizados, mas é o completo o suficiente para executar uma pilha de software moderna. [3]

Com o propósito de implementar em VHDL (Vhsic Hardware Description Language) para FPGAs o modelo básico do de processador RISC-V de ciclo único descrito no capítulo 4 de Petterson e Hennesy (2018). Propõem-se a especificação de uma arquitetura que implementa a conjunto de instruções de inteiros de base 32bits (RV32I) , a qual será prototipada em FPGA (Field Programmable Gate Array).

1.1 OBJETIVO GERAL

O principal objetivo deste projeto consiste na descrição em VHDL de uma arquitetura do processador RISC-V de ciclo único, que implementa uma versão reduzida do conjunto de instruções da ISA (RV32I). A implementação proposta será validada por simulação e prototipação em FPGA utilizando o kit de desenvolvimento Altera Cyclone IV com chip-on-bord EP4CE622C8N.

1.2OBJETIVOS ESPECÍFICOS

O projeto tem como objetivos específicos:

- Estudo das instruções da base I do RISC-V de Petterson e Hennesy (2018), determinando as quais serão implementadas;
- Projeto e descrição em VHDL de um caminho de dados monocíclico;
- Projeto e descrição em VHDL dos dispositivos de memória utilizados para armazenamento de programa e dados;
- Projeto e descrição em VHDL de dispositivos periféricos (E/S) para interagir com o núcleo implementado;

.

1.3 JUSTIFICATIVA.

Em sistemas embarcados a utilização de hardwares com tamanho reduzido levam em consideração tanto o núcleo principal, quanto a memória quando se fala de processadores. Tais aspectos, buscam a redução do consumo máximo de energia, a fim de que “sistemas portáteis” tenham maior duração possível em completo funcionamento.

A ISA RISC-V propões como um de seus objetivos a redução do tamanho de código a ser executado pelo processador. Consequentemente há uma redução de tamanho de software reflete na possibilidade de se utilizar memórias menores, já que são um dos componentes de maior dimensão dentro dos sistemas.

2 PESQUISA DE BASE

Neste capítulo se realiza a revisão de conceitos importantes para a compreensão de projeto o processador alvo. A seção 3.1 trata-se da apresentação da arquitetura e do conjunto de instruções utilizadas no projeto, o RISC-V. Na seção 3.2 é realizada uma abordagem geral sobre a arquitetura e organização de computadores, onde um dos pontos a serem destacados é a descrição dos características do design do processador RISC-V monocíclico que será implementado.

De acordo dom Petterson e Mennessy (2017) está regra de se manter exatamente três operando estende-se para as demais operações aritméticas, e está de acordo com a filosofia de manter um hardware simples, uma vez que um número variável de operandos exigiria uma maior complexidade de hardware. Este caso ilustra o "Princípio de Design 1: Simplicidade favorece a regularidade".

2.1 RISC-V ISA

O RISC-V originalmente desenvolvido na Divisão de Ciência de Computação da Universidade da Califórnia, em Berkeley, é uma arquitetura de Califórnia de instruções (ISA) livre, aberta e extensível que forma base de diversos projetos de processadores programáveis que estão sendo implementados na última década. Em contraste aos esforços anteriores que projetaram núcleos de processadores abertos, RISC-V é uma especificação ISA destinada a permitir que muitas implementações de hardware distintas alavanquem o desenvolvimento de software comum. Os benefícios de um padrão aberto são múltiplos, mas talvez o mais importante seja a abundância potencial de implementações do processador. Implementações gratuitas e de código aberto reduzirão o custo da criação de novos sistemas. [2] De acordo com a RISC-V Foundation (2019), a qual é formada atualmente por mais 235 membros, afirma-se que esta iniciativa de liberdade de software abre caminho para os próximos 50 anos de design e inovação em computação.

O RISC-V é projetado como uma arquitetura modular com variantes cobrindo espaços de endereço de 32 bits, 64 bits e 128 bits. O conjunto de instruções de base inteira é enxuto, exigindo menos de 50 níveis de usuário instruções de hardware para suportar uma pilha de

software moderna completa, permitindo que os projetistas de microprocessadores criem rapidamente protótipos totalmente funcionais e adicionem recursos adicionais de forma incremental. [3]

2.1.1 O conjunto de instruções RV32I

RV32I é m conjunto de instruções simples de base inteira , que conta com 47 instruções, de 32 bits de tamanho, tendo 6 formatos diferenciados pelo 7 bits menos significativos chamados de código de operação ou OPCODE .O RV32 é um Load/Store ISA, significando que os valores de dados devem ser trazido para o Arquivo de Registro antes de serem operados

No RV32I existem 31 registradores que são de uso geral representados de X1 à X31 com a respectiva largura de 32bits. O RISC-V dedica O registro n° 0 ou X0 a ser conectado ao valor zero.. [5] Além dos 32 registradores mencionados, existe um registrador de uso específico nomeado Program Counter (PC) [2]. A Figura (1) demonstra o estado da arquitetura visível ao usuário de RV32I.

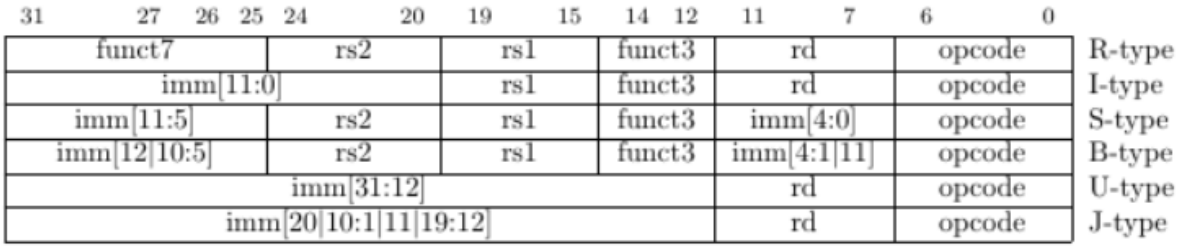
Figura 1: Registradores do RV32I

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Fonte: [2]

. O espaço de memória do RV32I consiste em 2^{32} locais (o que significa que tem um espaço de endereçamento de 32bits) e cada local contém 8bits (o que significa que o RV32 tem uma endereçabilidade por byte). O controle de programa é mantido pelo contador de programas(PC). O PC é um registro de 32bits que contém o endereço da instrução atual sendo executada.

Figura 2: Formatos de instruções do RV32I



Fonte: [2]

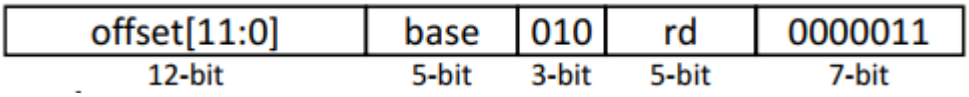
2.1.2 Instruções de memória

Instruções de movimentação de dados são usadas para transferir valores entre o arquivo de registros e o sistema de memória. A carga instrução *Load Word (Lw)* lê um valor de 32bits da memória e coloca em um registrador. A instrução de armazenamento *Store Word (Sw)* obtém um valor de um registrador e grava na memória. O formato de instrução de *Load* tem código de operação de dos bits de instrução *Lw* [6:0] é 0000011 . O endereço efetivo (o endereço do local de memória a ser lido) é especificado pelos campos rs1 e pelo imm [11:0]. O endereço é calculado adicionando o conteúdo do rs1 ao campo imm de sinal estendido.

- Assembly:

$$Lw\ rd, offset_{12}(base)$$

- Código de máquina:



- Semântica:

$$Byte_{adress_{32}} = signExtend(offset_{12}) + Rs1(base)$$

$$Rd <- Mem_{32}[byte\_addres\ ]$$

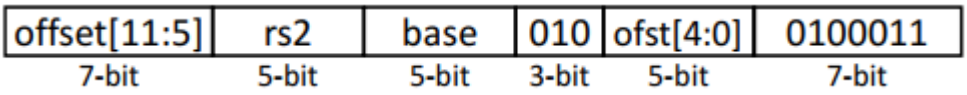
- Variações: LW, LH , LHU, LB, LBU.

O formato da instrução de armazenamento, SW, é mostrado abaixo. O código de operação desta instrução é 0100011. Assim como com a carga instrução (LW), o endereço efetivo é a localização da memória especificada por rs1 e imm [11: 0]. O endereço é formado da mesma maneira que o LW, exceto que os bits de deslocamento imm [4: 0] vêm da parte rd de a instrução em vez da porção rs2. Isso é para garantir que os sinais para selecionar qual índice de registro ler ou escrever não são dependentes de qual é a opcode da instrução.

- Assembly:

$$Sw\ rs2, offset_{12}(base)$$

- Código de máquina:



- Semântica:

$$Byte_{addres_{32}} = signExtend(offset_{12}) + Registro(base)$$

$$MEM_{32}[byte_{addres}] <- Rs2$$

- Variações : SW, SH, SB

2.1.3 Instruções aritméticas

O RV32I é capaz de realizar 10 operações aritméticas com registradores para instruções do Tipo-R e 9 operações de inteiros imediatos, para instruções do Tipo-I. As instruções da Figura 3 representam adição, subtração, deslocamento lógico à esquerda, comparação de menor que (assinalada), comparação de conjunto menor que (não assinalada), disjunção exclusiva bit a bit, deslocamento lógico à direita, disjunção bit a bit e conjunção bit a bit. O código de operação para instrução Tipo-R é 0110011 e para Tipo-I é 0010011. As operações são decodificadas a partir dos campos Func3 [14:12] e bit [30] para identificar a subtração e outras instruções que compartilhas do mesmo Func3.

Figura 3 Instruções aritméticas

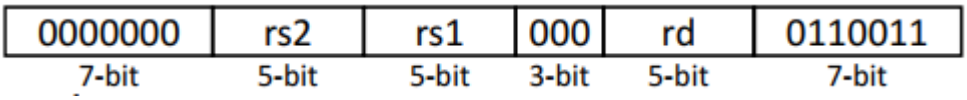
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Fonte: [4].

- Assembly(Tipo-R) :

$$Add\ rd, rs1, rs2$$

- Código de máquina:



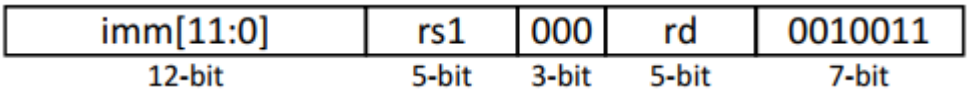
- Semântica:

$$RD < -Rs1 + Rs1$$

- Assembly(Tipo-I):

$$Addi\ rs, rs1, imm_{12}$$

- Código de máquina:



- Semântica:

$$Rd < -Rs1 + sigExtend(imm)$$

- Exceções: Para instruções do tipo I não é permitido fazer subtração.

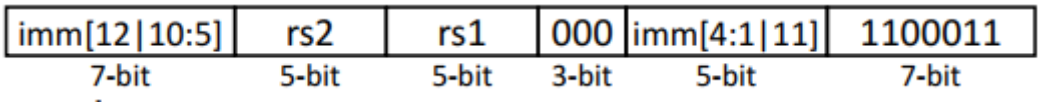
2.1.4 Instruções de Controle

As instruções de transferência condicional de fluxo *Branchs* de acordo com o resultado da comparação. Elas fazem com que o controle do programa se ramifique para um endereço se a relação entre o primeiro operando e o segundo operando for igual, diferente, menor que ou maior que. Quando o fluxo é levado, o endereço da próxima instrução a ser executada é calculada somando o valor do PC atual ao campo imediato do formato tipo SB.

- Assembly

$$Beq\ Rs1, Rs2, imm_{13}$$

- Código de máquina



- Semântica:

$$Pc_{next} = PC_{curret} + singExtend(imm_{13})$$

$$se\ Rs1 = Rs2$$

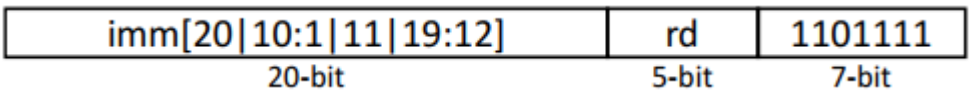
- Variações : BEQ, BNE, BLT, BGE

As instruções de controle incondicional JUMP na Link

- Assembly :

$$JAL\ Rd\ imm_{21}$$

- Código de máquina :



- Semântica:

$$Pc_{next} = Pc_{current} + signExtend(imm_{21})$$

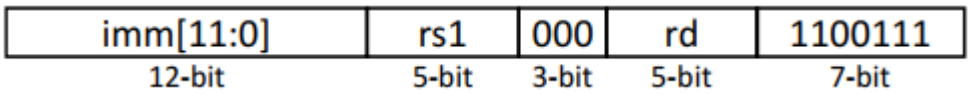
$$Rd < -Pc + 4$$

Instrução Jump Indirect

- Assembly :

$$JALR\ Rd, Rs1, Imm_{12}$$

- Código de máquina



- Semântica :

$$Pc = Rs1 + signExtend(Imm_{12})$$

$$Rd < -Pc + 4$$

2.2 ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Sob uma ótica de alto nível, um dispositivo computacional consiste em uma CPU (Unidade Central de Processamento), Memória, e dispositivos de entrada e saída conectados através de barramentos para transferência de dados ou módulos de comunicação entre componentes. A CPU é responsável pelo controle das operações em geral processando os dados, a memória principal é utilizada para armazenamento de dados , e os dispositivos de entrada e saída estabelecem as comunicações externas do processador. [6]

O processador trabalha em geral realizando operações descritas no conjunto de instruções de máquina ISA. Para isso é necessário componentes como:

- Banco de Registradores – memória interna do núcleo processador pequena e rápida, utilizada para armazenar dados em uso;

- Unidade Lógica Aritimética (ALU – Aritimetical Logic Unit) – realiza a execução das operações lógicas aritiméticas como subtrações, adições, comparações e lógica AND e OR.
- Contador de Programa (Program Counter – PC) Registrador que aponta para a instrução da memória a ser buscada;
- Unidade de Decodificação – Decodifica os campos da instrução que correspondem a instrução.
- Registradores temporários – Registradores utilizados para armazenar dados com a instrução lida da memória, os endereços lidos do banco de registradores, ou então resultados da ALU.

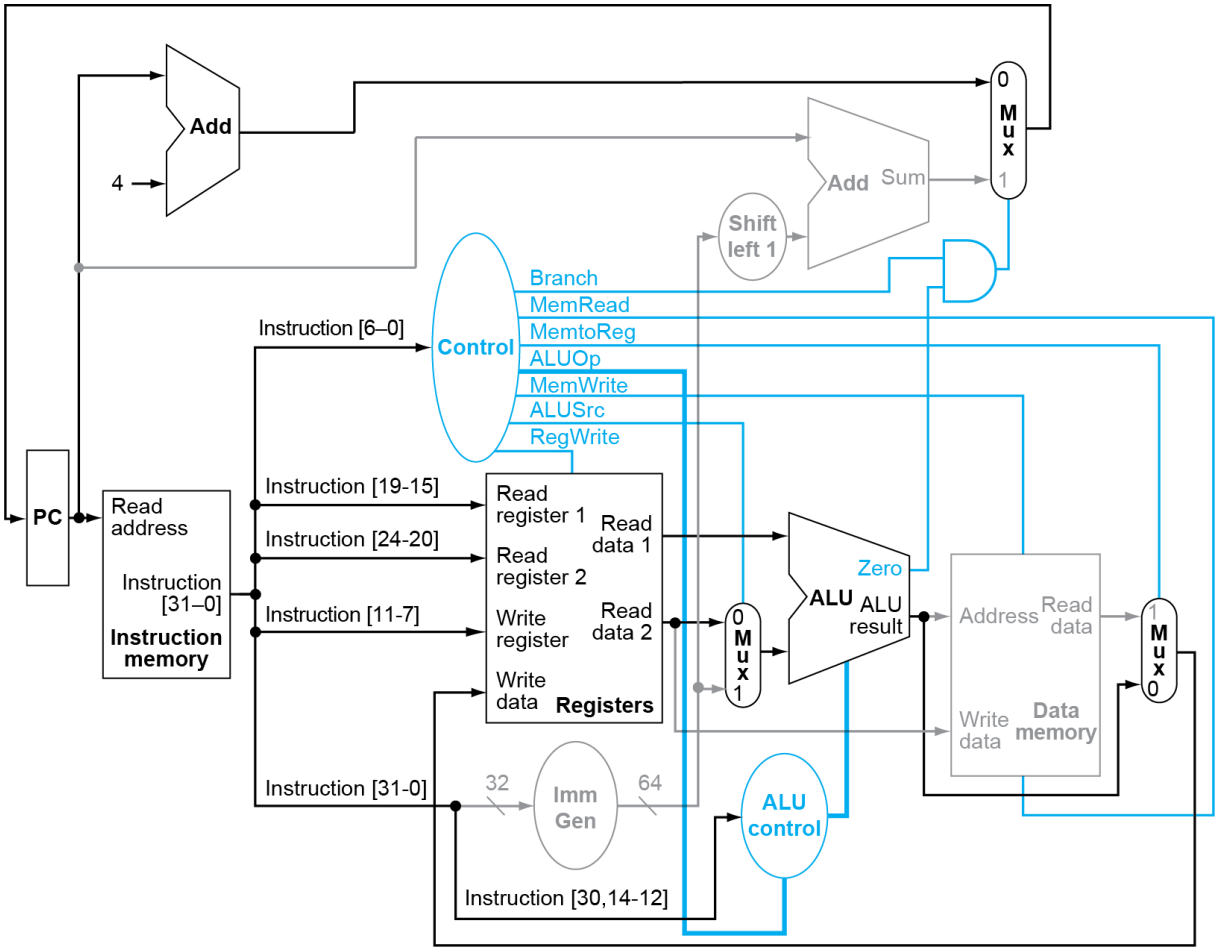
2.2.1 PROCESSADOR MONOCÍCLO

Em núcleos monocíclicos tem-se como exemplo o caminho de dados na Figura () do processador RISC-V já com o bloco de controle, nele o fluxo de dados ocorre da esquerda para direita , iniciando pelo PC (Program Counter), que aponta para o endereço da memória da instrução a ser acessado, em paralelo ocorre o incremento de PC para apontar a instrução que será buscada e executada no próximo ciclo de *clock*. Após obter a instrução, seus campos são separados na decodificação para as diferentes unidades, os campos que detém o código de operação (*opcode*) vai para a unidade de controle (control) que correspondem aos bits 0 à 6, outros campos são conectados com banco de registradores indicando os endereços a serem lidos (*rs1 bits 19 a 15 e rs2 bits 24 a 20*) e o da escrita (*rd bits 11-7*). E por fim os campos do sinal imediato. O restante da operação do núcleo depende da decodificação da instrução que consiste de um circuito combinacional, se for uma instrução do tipo aritimética/lógica, por exemplo, se utilizaria a ALU para executar a operação desejada, e então se armazenaria o resultado de volta no banco de registradores. Porém se fosse uma instrução de transferência de dados, como a *ld*, se calcularia endereço de acesso na ALU e então o utilizaria para fazer a leitura da memória de dados, para no final armazenar o valor no banco de registradores. Os detalhes da execução das instruções ao longo do caminho de dados são descritas por Petterson e Hennessy (2017 cpt. 4). Percebe-se que na abordagem monocíclica as instruções podem percorrer muitos caminhos diferentes dentro do hardware ao longo de sua execução, e por fim possuem tempos de execução

distintos, por isso, é necessário que o período do sinal de *clock* seja o grande suficiente para acomodar a mais lentas instruções.

A Figura (4) ilustra o caminho de dados monocíclico, onde os bits 6-0 da instrução correspondem a operação da instrução, os bits 19-15, 24-20 e 11-7 aos campos de leitura dos registradores 1 e 2 do banco de registradores, e endereço de escrita no banco de registros.

Figura: 4 Caminho de dados do processador RISC-V monocíclico.



Fonte: [5].

3 MÉTODOLOGIA

Este capítulo está descrito a metodologia seguida bem como as especificações definidas para o desenvolvimento do processador, que possui a implementação de um núcleo monocíclico. A primeira leitura realizada foi a do manual de instruções RISC-V editado por Waterman e Asanovic (2017) e também os capítulos 2 da obra de Petterson e Hennessy (2017) entre outros documentos a fim de se familiarizar com a ISA, e determinar as especificações iniciais de projeto.

Após a revisão da ISA , o próximo passo foi o estudo do caminho de dados apresentado por Petterson e Hennessy (2017 cpt. 4), do qual e tido como base para este projeto.

Após finalizado o projeto do núcleo monocíclico, iniciou-se a implementação em VHDL da arquitetura, iniciando pela descrição de hardware individual do bloco de dados da unidade de processamento, seguido por testes realizados em simulações no software Modelsim.

O próximo passo foi a implementação da unidade de controle do caminho de dados unido a mesma ao bloco de dados.

Em continuidade, elaborou-se uma estratégia para realizar as estradas e saídas de dados na unidade de processamento com dispositivos de interação endereçados na memória.

A validação do projeto é realizada através de Testbenchs elaborados no software Modelsim, e a execução de programas testes.

Fonte: O Autor (2018).

3.1 ESPECIFICAÇÕES INICIAIS

Algumas definições do projeto e pré-requisitos devem ser atendidas conforme o conjunto de instruções a ser implementado do modelo básico de RISC-V descrito em Petterson e Hennessy (2017 cpt4 pg.)

1) Ele deve ser descrito em VHDL, implementando o conjunto de instruções de inteiros de base, 32 bits (RV32I), visando a síntese de FPGA.

2) A operação *multi* deve ser executada o mais rápido possível, para não atrasar a arquitetura de ciclo único proposta.
3) Uma estratégia de E / S mapeada na memória deve ser definida e implementada. Os bytes 0H a 7H da memória de dados devem ser usados para operações de saída e os bytes 8H a FH devem ser usados para operações de entrada.

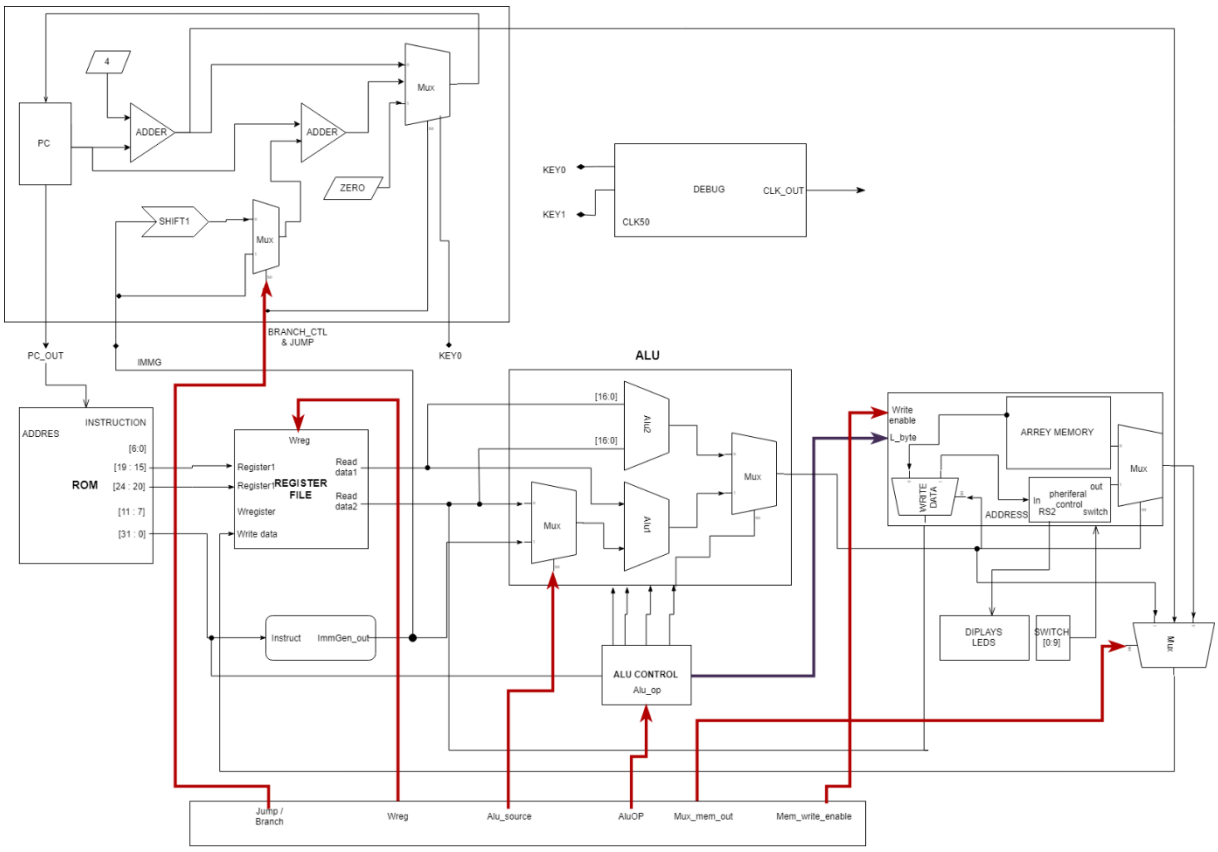
4) A arquitetura deve ser capaz de executar as seguintes instruções:

- Addition (with overflow) – add rd, rs1, rs2
- Addition immediate (with overflow) – addi rd, rs1, imm
- Subtract (with overflow) – sub rd, rs1, rs2
- AND – and rd, rs1, rs2
- AND immediate – andi rd, rs1, imm
- XOR – xor rd, rs1, rs2
- XOR immediate – xori rd, rs1, imm
- OR – or rd, rs1, rs2
- OR immediate – ori rd, rs1, imm
- Shift left logical – sll rd, rs1, rs2
- Shift right logical – srl rd, rs1, rs2
- Set less than – slt rd, rs1, rs2
- Branch on equal – beq rs1, rs2, offset
- Branch on not equal – bne rs1, rs2, offset
- Branch less than – blt rs1, rs2, offset
- Branch greater than equal – bge rs1, rs2, offset
- Jump and link – jal rd, offset
- Jump and link register – jalr rd, rs1, offset
- Load byte – lb rd, offset(rs1)
- Load word – lw rd, offset(rs1)
- Store byte – sb rs2, offset(rs1)
- Store word – sw rs2, offset(rs1)

4 ESPECIFICAÇÕES DO DESIGN

Neste capitulo é realizada a descrição de cada bloco implementado em VHDL com as suas respectivas funções. A Figura () apresenta a arquitetura RISC-V projetada.

Figura 5: Esquemático da arquitetura do processador RISC-V implementado.



Fonte: O Autor (2019).

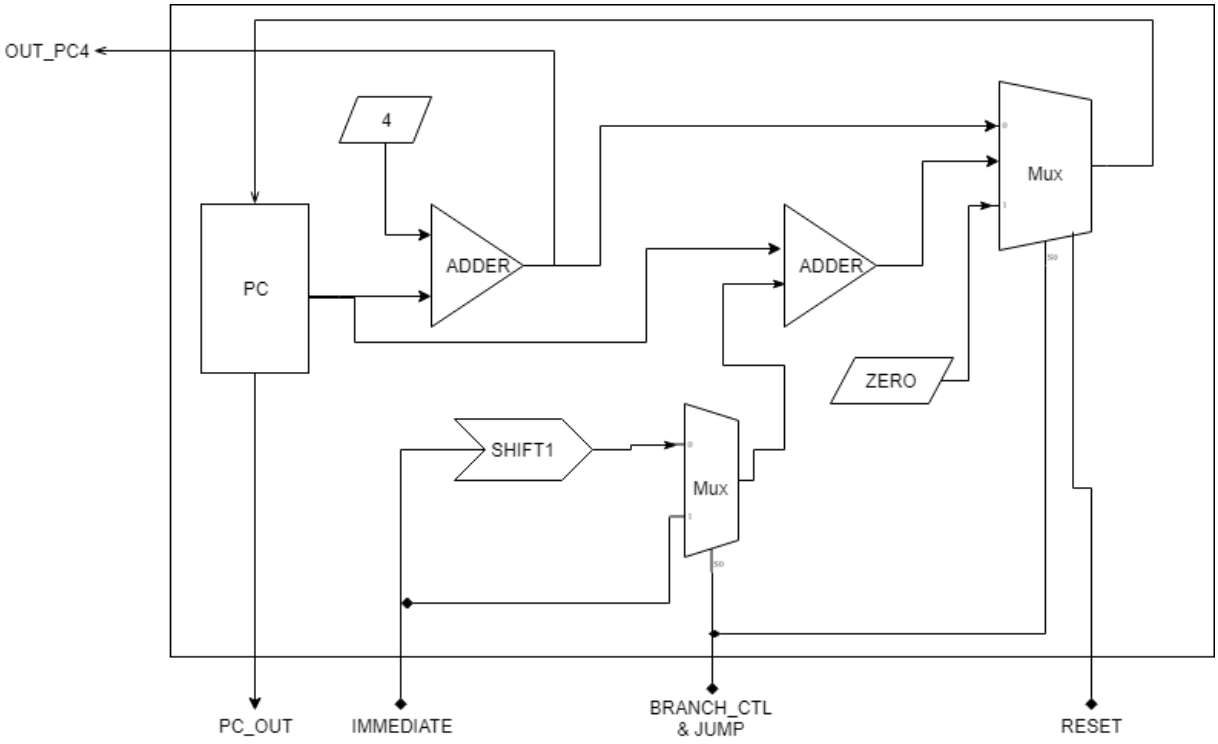
4.1 PROGRAM COUNTER

O PC (Program Counter), aponta para o endereço da memória da instrução a ser executada. Em paralelo ocorre o incremento de PC para apontar a instrução que será buscada e executada no próximo ciclo de *clock*. O PC pode receber quatro tipos de incremento condicional, e zero quando ocorre um reset.:

- PC_next = PC+4 para instruções no formato R, I(exceto o Jalr) e S
- PC_next = PC+Immediate(13)(shift 1 <<) para instruções no formato SB;
- PC_nex = PC+Immediate(21) para a instrução Jump and Link;
- PC_next = Reg(Rs1) + Immediate(12) para a instrução Jump and Link Register.

Nas instruções de Salto incondicional é salvo no Rd o valor de PC+4. A Figura (6) mostra o esquemático do componente projetado.

Figura 6: Esquemático do Program Couter



Fonte: O Autor (2019).

4.2 MEMÓRIA DE INSTRUÇÃO

A memória ROM (*Read Only Memory*) ou memória de instrução implementada possui uma porta de leitura de largura de 32bits, e nenhuma porta de escrita, pois a mesma não é reprogramável durante a execução do programa. O componente tem uma porta de entrada que recebe o endereço de 32bits da instrução a ser executada informado pelo PC (*Processor Counter*). Internamente as instruções são armazenadas em bytes, portanto cada instrução é lida de 4 em 4 bytes.

Os endereços da memória ROM conta com 32 linhas de endereços sendo 2^{32}e células de memória e 8 linhas de dados (célula da memória constituída de 8bits), logo a capacidade total é 4MBytes.

Na mesma entidade VHDL da memória ROM implementou-se um decodificador para separar os campos das instruções lidas na memória como pode ser observado na Figura(7)

Figura 7: Trecho de código VHDL do decodificador

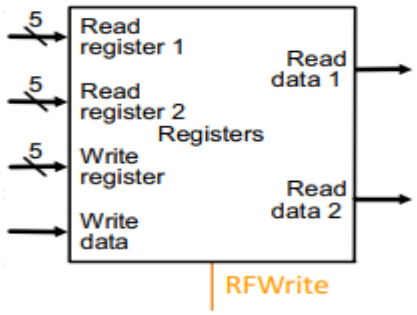
```
decoder_instruction: process(out_inst)
begin

instruct01<= out_inst(31 downto  0);
instruct02<=out_inst(31 downto  0);
opcode  <= out_inst(6  downto  0);
rd      <= out_inst(11 downto  7);
rs1     <= out_inst(19 downto 15);
rs2     <= out_inst(24 downto 20);

end process decoder_instruction;
```

Fonte: Quartus II.

4.3 BANCO DE REGISTRADORES

De acordo com a base de inteiros adotada nas especificações de projeto, o banco de registradores representado no bloco da Figura (8) é do tamanho 32x32, 32 registradores de 32 bits, possui duas portas de entrada de leitura dos endereços dos registros de origem *rs1* e *rs2*, duas para leitura de dados dos registros, duas entradas de escrita de endereço de destino *rd* e uma para o dado, e uma porta que receberá o sinal de controle para a ação de escrita *RegWrite.*

Figura: 8 Banco de registrador



Fonte: O autor 2019

Quando o banco de registrador recebe os endereços de *rs1* e *rs2*, ele é atualizado na saída síncrona do *clock*. O registro de escrita só recebe o endereço de escrita caso a instrução necessite que seja escrito um valor no registro.

4.4 GERADOR DE IMEDIATO

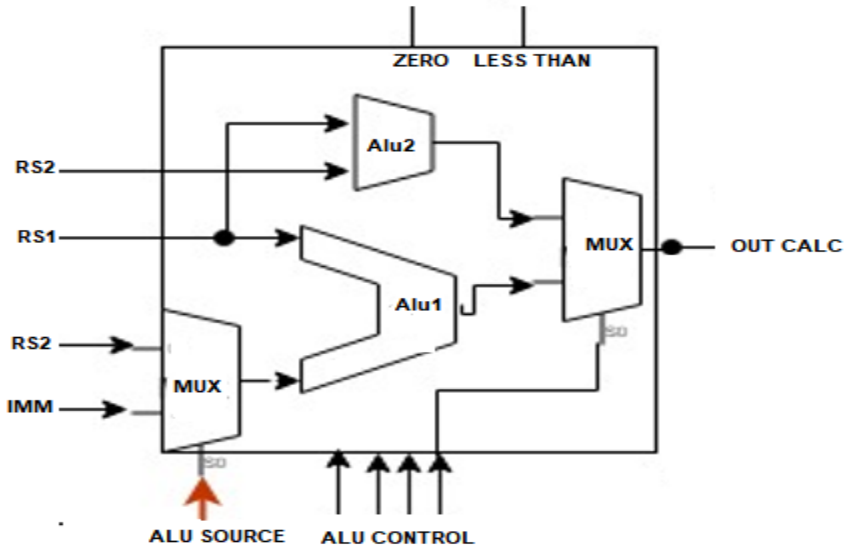
O Gerador Imediato recebe a instrução de 32 bits, e com base no opcode decide quais bits da instrução devem ser interpretados como o campo imediatos. Então colocar o sinal e estender os bits imediatos para 32bits. A parte opcode da instrução é bit [6 : 0]. Para o conjunto de instruções do Tipo-I o opcode é "0010011" para operações aritméticas ou 0000011 para instruções de carga na memória e também para a instrução JARL cuja o opcode é "1100111" e todas possuiem o campo do imediato de 12bits [31: 20]. Para o conjunto de instrução do Tipo-S o opcode é "0100011"e o campo de imediato de é decodificado nos bits [31:25][11:7]. Para o conjunto de instruções do Tipo-SB o opcode é "1100011" e o campo de imediato é decodificado nos bits [31][7][30:25][11:8]. A instrução JAL possui opcode "1101111" e o campo de imediato decodificado nos bits [31][19:12][20][30:25][24:21]&[0]. Para as instruções que não necessitam do campo de imediato é atribuido zeros em todos os 32 bits de saída.

4.5 UNIDADE LÓGICA ARITMÉTICA

A Unidade Lógica Aritmética implementada é formada por duas entidades denominadas ALU1 e ALU2. A ALU1 possui 2 operandos de 32 bits sendo que um deles o registrador (RS1) e o outro de uma saída multiplexada entre um registrador (RS2) e o campo

de Imediato (ImmG). A ALU 1 realiza operações de add, sub, srl, sll, slt,and, or, xor e também possui um bloco para testas as condições de responsável por fazer os testes de branchs e alterar o nível lógico das flags(zero e less than). A ALU2 é um multiplicador de 16x16 bits entre dois registradores. A saída das duas ALUs estão ligadas á um multiplexador que tem o seu select controlado pela *ALU control* assim como todas as demais operações realizadas na ALU.

Figura: 9 Bloco lógico da ALU



Fonte: O Autor (2019).

O Multiplicador implementado representado na Figura (9) como Alu2 trabalha com operandos de 16 bits e apresenta como produto o resultado em 32bits. Um dos grandes problemas seria o número de ciclo de clock que o multiplicados iria levar para realizar a operação. As operações são realizadas em paralelo a fim de realiza-las da forma mais rápida possível para não atrasar o *clock* de ciclo único. Cada bit mais significativo do operando multiplicador e o produto é deslocado para direita n vezes a posição so bit multiplicador. Após cada operação dessa são realizadas somas consecutivas formando o produto resultante da multiplicação.

4.6 ALU CONTROL

A Unidade de controle de operações lógicas e aritméticas ALU CONTROL fornece para a ALU os sinais de controle principalmente com base na decodificação de uma seleção de bits da instrução. É decodificado também o tipo de condição (*Branch*) que será testada na ALU. Outra instrução decodificada é a Load/Store Byte, quando executada, uma flag é sinalizada levando um pino para nível logico alto. Este pino é responsável por controlar a largura do dado que será armazenado ou carregado da memória. Na tabela (xx) estão descritos os bits de decodificação e os sinais de saída.

Tabela 1: Sinais de saída da ALU control e as respectivas operações.

ALU_OP	ALU_CTL_in [25][30][14:12]	ALU_CTL_out	Operação	L_Byte
10	00000	0000	add	-
10	01000	0001	Sub	-
10	00111	0010	And	-
10	00110	0011	Or	-
10	00100	0100	Xor	-
10	00001	0101	Sll	-
10	00101	0110	Slr	-
10	10000	0111	Mult	-
10	10111`	0010	Andi	-
10	10110	0011	Ori	-
10	10100	0100	Xori	-
10	00010	1000	Slt	-
00	xx010	0000	Add	0
00	xx000	0000	Add	1
01	xx000	1001	Beq	-
01	xx001	1010	Bne	-
01	xx100	1011	Blt	-
01	xx101	1100	Bge	-

Fonte: O Autor (2019).

4.7 MEMÓRIA RAM

O armazenamento das variáveis e estruturas de dados dos programas são armazenados na memória de dados, uma memória RAM de 2Kbytes. O processador RISC-V possui apenas uma interface de entrada e saída para memória de dados externa. Como na arquitetura RISC-V não existem instruções dedicadas a entrada e saída de dados para periféricos, o acesso do processador aos registradores dos periféricos é feito através da interface de memória. Portanto, no mapeamento da memória foi dedicado os endereços 0x00 á x03 para escrita nos periféricos e 0x04 a 0x07 para leitura dos periféricos. Então a memória de dados só pode ser escrita ou lida a partir do endereço 0x08. Na seção(XX) abordados os endereços específicos de cada periférico O processo de escrita/ leitura é controlado por um sinal comandado pela unidade de controle principal, e a largura do dado que será escrito/lido é controlado pela Alu Control que disponibiliza de uma flag que identifica se o dado e uma palavra de 32 bits ou palavra de 8 bits.

4.8 MULTIPLEXADOR MEM_TO_REG

Foi implementado um multiplexador onde a sua saída está diretamente ligada ao banco de registrador. As entradas e o respectivo sinal de controle estão descritos na Tabela(2)

Tabela 2 : Tabela verdade do multiplexador Mem_to_Reg.

Mem_to reg	Out_to_reg
00	ALU_result
01	Mem_out
10	PC+4
xx	0

Fonte: O Autor (2019).

A entrada Alu result é selecionada quando há uma instrução do tipo R ou operação aritmética com imediato. A entrada Mem_out é selecionada quando há um a instrução de carga

da memória, e a entrada PC+4 é selecionado quando ocorre uma instrução de salto quando é necessário salvar o valor de PC+4 no banco de registrador.

4.9 BRANCH CONTROL

A Unidade de controle para as instruções *branch* tem como entrada um sinal vindo da Unidade de controle principal informando que se trata de uma branch e Também recebe um sinal da ALU que faz o teste condicional. O circuito lógico tem sua saída em nível lógico alto quando o corre uma branch.

4.10 UNIDADE DE CONTROLE PRINCIPAL

A Unidade de controle a ser projetada recebe os bits [6:0] da instrução referente ao *opcode*. São decodificados 7 *opcodes* diferentes identificando o tipo da instrução e setados os bit de saída que controlam o caminho de dados do *datapath.* O Quadro (1)representa a tabela verdade do funcionamento da Unidade de controle.

Quadro 1: Decodificador do opcode.

Type	**Instruction**	**Opcode**	**Aluop**	**Alu source**	**Wreg**	**Branch**	**Mux_mem out**	**Mem_write en**	**Jump**
R-Type	And,or,xor Sll,srl,slt,mul Add,sub	0110011	10	0	1	0	00	0	0
I-Type	Addi,andi,ori Slli,xori	0010011	10	1	1	0	00	0	0
I-Type	Lw,Lb	0000011	10	1	1	0	01	0	0
I-Type	JalR	1100111	00	0	0	0	00	0	1
J-Type	Jal	1101111	00	0	1	0	10	0	1
SB-Type	Beq,bne,blt,bge	1100011	01	0	0	1	00	0	0
S-Type	Sw,sb	0100011	00	1	0	0	00	1	0

Fonte: O Autor (2019).

Tabela 3 : Sinais de controle e suas funções.

Sinal	valor	função
Alu_op	10	Alu control decodifica intruções Tipo-R
	00	Alu control decodifica instruções Load e Store
	01	Alu Control decodifica o tipo de branch
Alu_source	0	Alu recebe o Rs2 como 2° operando
	1	Alu recebe o Imm como 2°operando
Wreg	0	Desabilita a escrita no Register File
	1	Habilita a escrita no Register File
Branch	0	-
	1	Sinaliza uma branch
Mux_mem_out	00	Wregister recebe Alu
	01	Wregister recebe dado da memória
	10	Wregister recebe PC+4
Mem_write_en	0	Leitura da memória
	1	Escrita da memória
Jump	0	-
	1	Sinaliza um Jump

Fonte: O Autor (2019).

4.11 PERIFÉRICOS

O controlador de periféricos foi implementado na mesma entidade da memória. Como a mesma é mapeada foi destinado os seguintes endereços

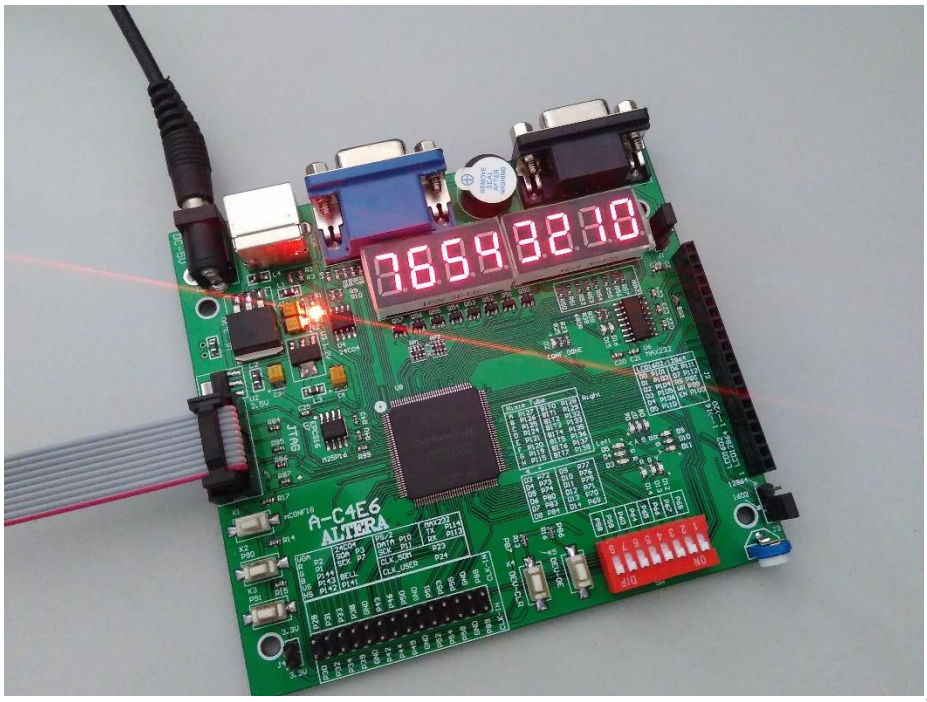
Tabela 4: Periféricos e seus endereços na memória mapeada.

Nome dispositivo	Endereço na memória	Instrução
Switchs[0:9]	0x04	Load
Leds [0:9]	0x00	Store
Display[0:1]	0x01	Store
Display[2:3]	0x02	Store
Display[4:5]	0x03	Store

Fonte: O Autor(2019).

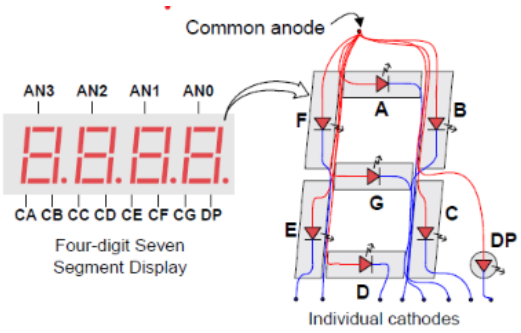
O kit de ALTERA Cyclone IV FPGA Board A-C4E6 da Figura(10) disponibiliza de dois displays de 4 dígitos anodo comum, 12 leds, 8 chaves DIP, 4 push bottons.

Figura 10: Kit da ALTERA Cyclone IV FPGA Board A-C4E6



Fonte: O Autor (2019).

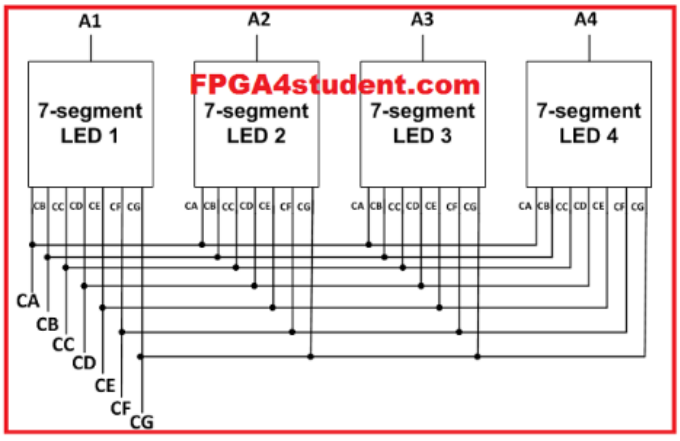
Figura 11: Display de 4 dígitos disponível no Kit.



Fonte [7].

Sete anodos dos sete segmentos s em um único LED são conectados juntos a um nó ânodo comum, enquanto seus cátodos são separados, como mostra a figura a seguir. O segmento DP é para iluminar o ponto então omitimos o segmento DP por enquanto, já que ele não está contribuindo para o valor numérico da exibição de sete segmentos . Então omitimos o segmento DP por enquanto, já que ele não está contribuindo para o valor numérico da exibição de sete segmentos . Os displays estão ligados todos em paralelo como mostra a Figura(12).

Figura 12: Ligação dos displays .



Fonte: [7]

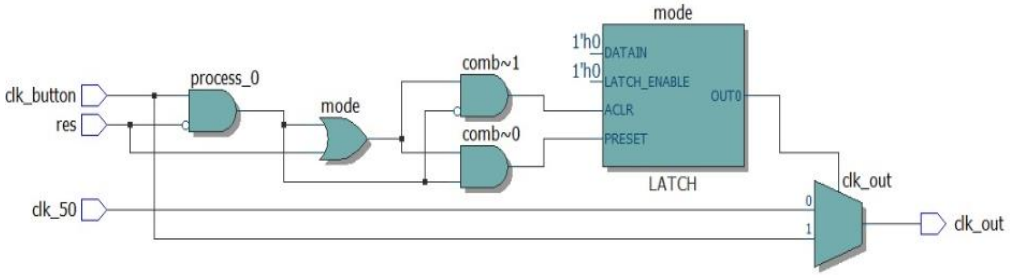
Assim, para exibir 4 números diferentes no display LED de quatro segmentos de quatro dígitos, temos que controlar os cátodos. Quando um LED é desativado após a iluminação, ele

escurece. Para evitar a descontinuidade de exibição percebida pelo olho humano, os quatro LEDs de sete segmentos devem ser continuamente atualizados em cerca de 1KHz a 60Hz ou devem ser atualizados a cada 1ms a 16ms.

4.12 MODO DEBUG

O componente gerado para a função modo debug apresentado na Figura(13) é responsável por controlar a fonte de clock do sistema. O sistema inicia em mono normal com o clock de 50MHz, quando a chave KEY1 é acionada o sistema entra em modo de depuração, então a fonte de clock passa ser o botão KEY1. Para que seja alterada novamente a fonte de clock é acionado o botão reset KEY0 que muda o estado lógico da variável *mode* armazenada no latch.

Figura 13 : Circuito lógico gerado para implementar o modo debug.



Fonte: Quartus II.

5 RESULTADOS E DISCUÇÕES

Neste capítulo são mostrados os resultados da síntese RTL da CPU que compõem a arquitetura RISC-V.

5.1 IMPLEMENTAÇÃO EM RTL

Os módulos que compõem a arquitetura proposta para o processador e seus periféricos descritos no capítulo 4, foram implementados na linguagem VHLD. A síntese para FPGA foi feita utilizando a ferramenta Quartus II 15.0 Web Edition da Altera. A simulação do processador foi feita utilizando o software ModelSim, da Mantor Graphics. A arquitetura proposta foi prototipada em FPGA da família Cyclone VI da Altera.

5.2 CYCLONE IV

A implementação para a Cyclone IV foi no dispositivo EP4CE6E22C8N que conta com 6272 blocos lógicos. Os resultados da síntese desta impementação são mostrados na Tabela

Tabela 5: Resultados da Síntese FPGA

Device: EP4CE6E22C8N Cyclone IV FPGA		
	Logic cells	**Logic Registers**
Alu	582	
Alu ctl	15	0
Rom	350	0
Ram	2020	490
Reg_file	2417	1024
ImmG	62	0
Mult	526	0
Control	113	0
Branch_ctl	1	0
Debug	8	4
DisplayControl	51	24
TOTAL	6145	1514
Otimizado	5589	
ÁREA	89%	

Fonte: O autor (2019).

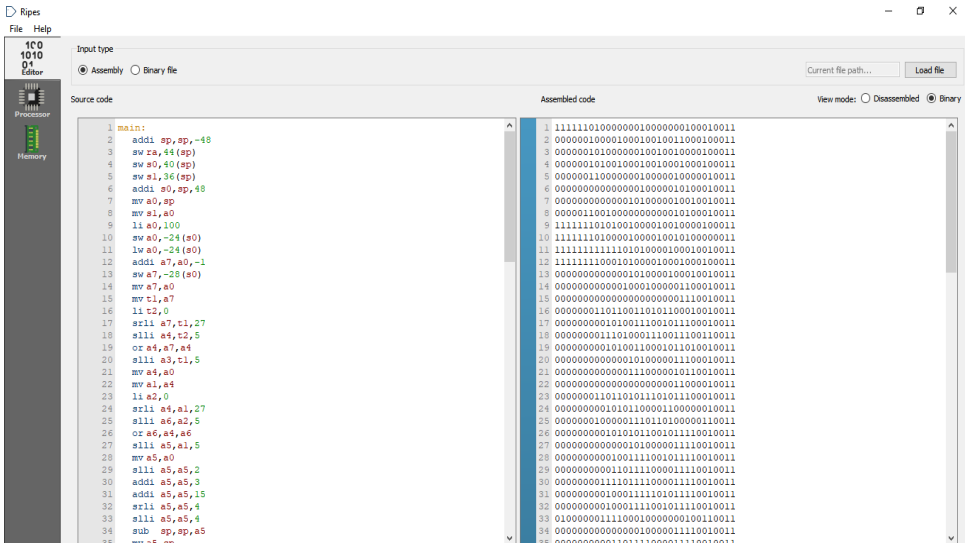
5.3 AMBIENTE DE VALIDAÇÃO

O ambiente de simulação e validação consiste no conjunto de descrições em RTL dos módulos desenvolvidos em linguagem VHDL e os programas desenvolvidos em assembly.

Através do software Modelsim foi possível visualizar o tempo de execução de cada instrução. Como é um processador Singl-Cycle foi possível ajustar a descrição de Hardware ao longo do desenvolvimento da arquitetura. Um dos aspectos que prejudicavam a execução em um ciclo era a inferência de latchs em alguns blocos lógicos, geralmente eles são criados quando é criado um processo combinacional ou atribuição condicional (em VHDL). Isso cria o que é conhecido como atribuição incompleta pelas ferramentas de síntese. A atribuição da saída não está completa em todas as possibilidades de entrada. Isso é ruim e deve ser evitado. Então a estratégia foi completar todas as atribuições.

Os programas *assembly* desenvolvidos para a validação foram montados em linguagem de máquina na plataforma Ripes mostrada na Figura(14)

Figura 14: Ripes - Montador de instrução RISC-V em linguagem de máquina.



Fonte: Ripes (2019).

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de uma plataforma simples de hardware, com processador com instruções de aplicação específica e periféricos. O hardware implementado é capaz de executar todas as instruções RISC-V de 32bits de base inteira. Foi implementado também um hardware um multiplicador, permitindo desta forma o acréscimo da instrução “mul” ao conjunto RV32I.

Pelo fato desta arquitetura ter a sua especificação aberta, tem despertado o interesse tanto do meio acadêmico quanto da indústria pelas suas características de simplicidade, mas principalmente pelo suporte à introdução de extensões à sua ISA com instruções específicas e customizadas para aplicação.

A Arquitetura apresentada como resultado do projeto é bastante adaptável o que permitirá a continuidade deste projeto inserindo mais extensões e melhorando o desempenho de hardware.

De modo geral a plataforma utiliza porcos recursos de hardware, incluindo memoria de dados e de programa ideal para utilização em FPGAs.

6.2 TRABALHOS FUTUROS

Este projeto atuou como um grande ponto de partida para a pesquisa do RISC-V. Como a arquitetura proposta pode ser estendida, o próximo passo será otimizar o design dos blocos implementados em VHDL, incluir as demais instruções de aplicação específica, adicionar outros periféricos.

O núcleo do processador RISC-V poderá também atuar como suporte à sistemas multicores com a adição de uma unidade de gerenciamento de memória, ou suporte à algum sistema operacional.

REFERÊNCIAS

1. FOUNDATION, R.-V. About the RISC-V ISA. **RISCV.org**, 2019. Disponivel em: <https://riscv.org/>. Acesso em: 20 maio 2019.
2. WATERMAN, A. **Design of the RISC-V Instruction Set Architecture**. University of California at Berkeley. Berkeley. 2016.
3. LEE, Y. et al. An Agile Approach to Building RISC-V Microprocessors. **IEEE Micro**, 36, 18 March 2016. 8-20. Disponivel em: <https://ieeexplore.ieee.org/abstract/document/7436635>. Acesso em: 21 maio 2019.
4. PETTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design RISC-V Edition:** The Hardware Software Interface. San Francisco: Morgan Kaufmann Publishers Inc., 2017. ISBN :0128122757 9780128122754.
5. ANDREW WATERMAN, K. A. **The RISC-V Instruction Set Manual**. 2.2. ed. Berkeley: [s.n.], v. 1, 2017. Disponivel em: <https://riscv.org/specifications/>.
6. STALLINGS, W. **Computer Organization and Architecture:** Designing for Performance. 8. ed. Upper Saddle River: Prentice Hall, 2010.
7. STUDENT, F. FPGA4 Student. **FPGA4 Student**. Disponivel em: <https://www.fpga4student.com/>.

APÊNDICE A

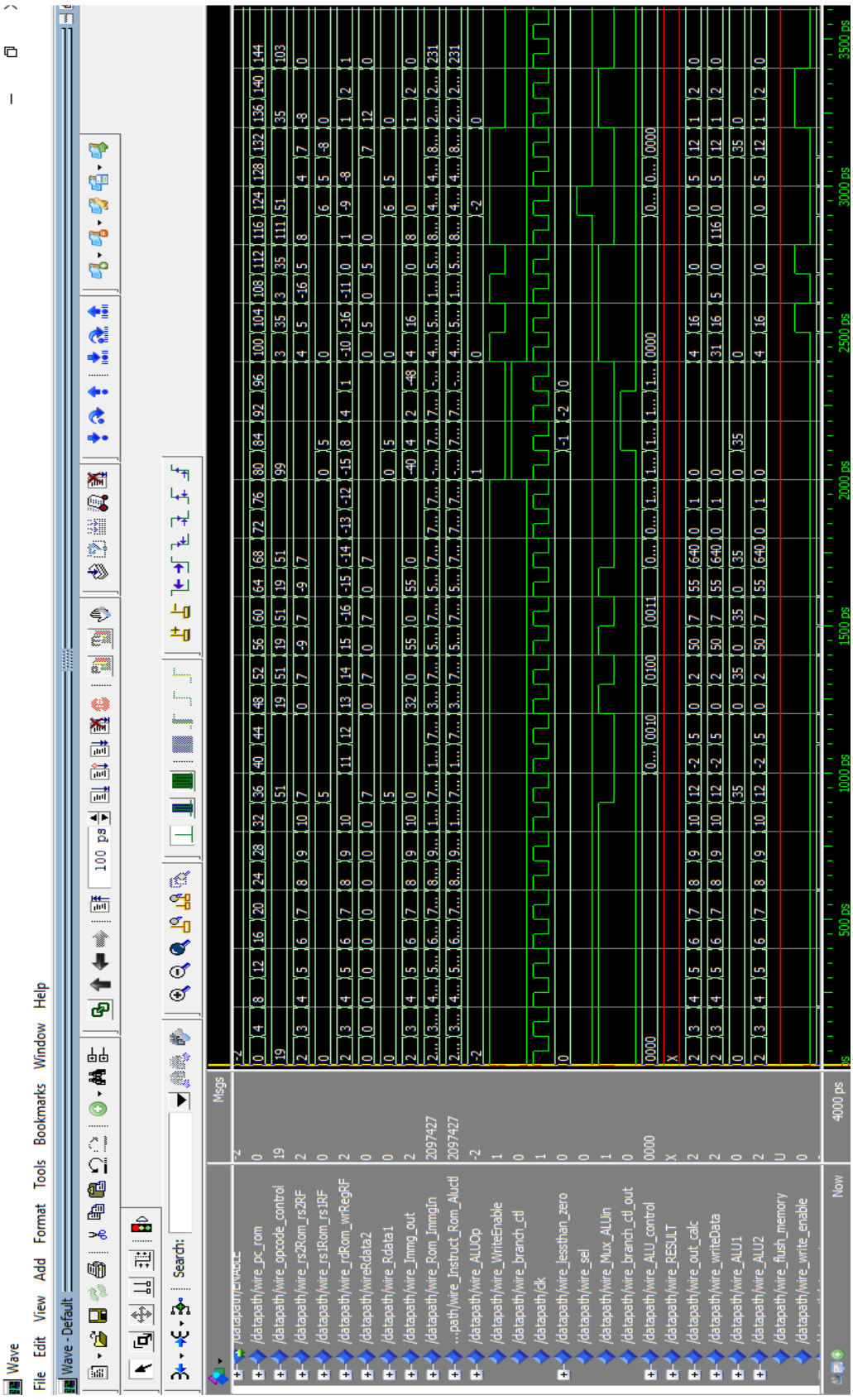
Programa teste com todas as instruções:

L1: addi x2,x0,2	"00000000","00100000","00000001","00010011",
addi x3,x0,3	"00000000","00110000","00000001","10010011",
addi x4,x0,4	"00000000","01000000","00000010","00010011",
addi x5,x0,5	"00000000","01010000","00000010","10010011",
addi x6,x0,6	"00000000","01100000","00000011","00010011",
addi x7,x0,7	"00000000","01110000","00000011","10010011",
addi x8,x0,8	"00000000","10000000","00000100","00010011",
addi x9,x0,9	"00000000","10010000","00000100","10010011",
addi x10,x0,10	"00000000","10100000","00000101","00010011",
add x10,x5,x7	"00000000","01110010","10000101","00110011",
sub x11,x5,x7	"01000000","01110010","10000101","10110011",
and x12,x5,x7	"00000000","01110010","11110110","00110011",
andi x13,x5,32	"00000010","00000010","11110110","10010011",
xor x14,x5,x7	"00000000","01110010","11000111","00110011",
xori x15,x5,55	"00000011","01110010","11000111","10010011",
or x16,x5,x7	"00000000","01110010","11101000","00110011",
ori x17,x5,55	
sll x18,x5,x7	
srl x19,x5,x7	
slt x20,x5,x7	
beq x0,x7,L1	
bne x5,x7,L2	
add x10,x5,x7	
L2: blt x5,x7,L4	

L4: bge x5,x7,L1	"11111010","01110010","11010000","11100011",
lb x22,4(x0)	"00000000","01000000","00001011","00000011",
sw x5,16(x0)	"00000000","01010000","00101000","00100011",
lw x21,16(x0)	"00000001","00000000","00101010","10000011",
sb x5,0(x0)	"00000000","01010000","00000000","00100011",
jal L5	"00000000","10000000","00000000","11101111",
add x0,x0,x0	"00000000","00000000","00000000","00110011",
L5: mul x23, x6,x8	"00000010","10000011","00001011","10110011",
sll x24, x5, x4	"00000000","01000010","10011100","00110011",
add x24, x24, x7	
sw x24, 1(x0)	
sw x24, 2(x0)	
jalr x0	

APÊNDICE B

Simulação RTL do programa 1 no ModelSim



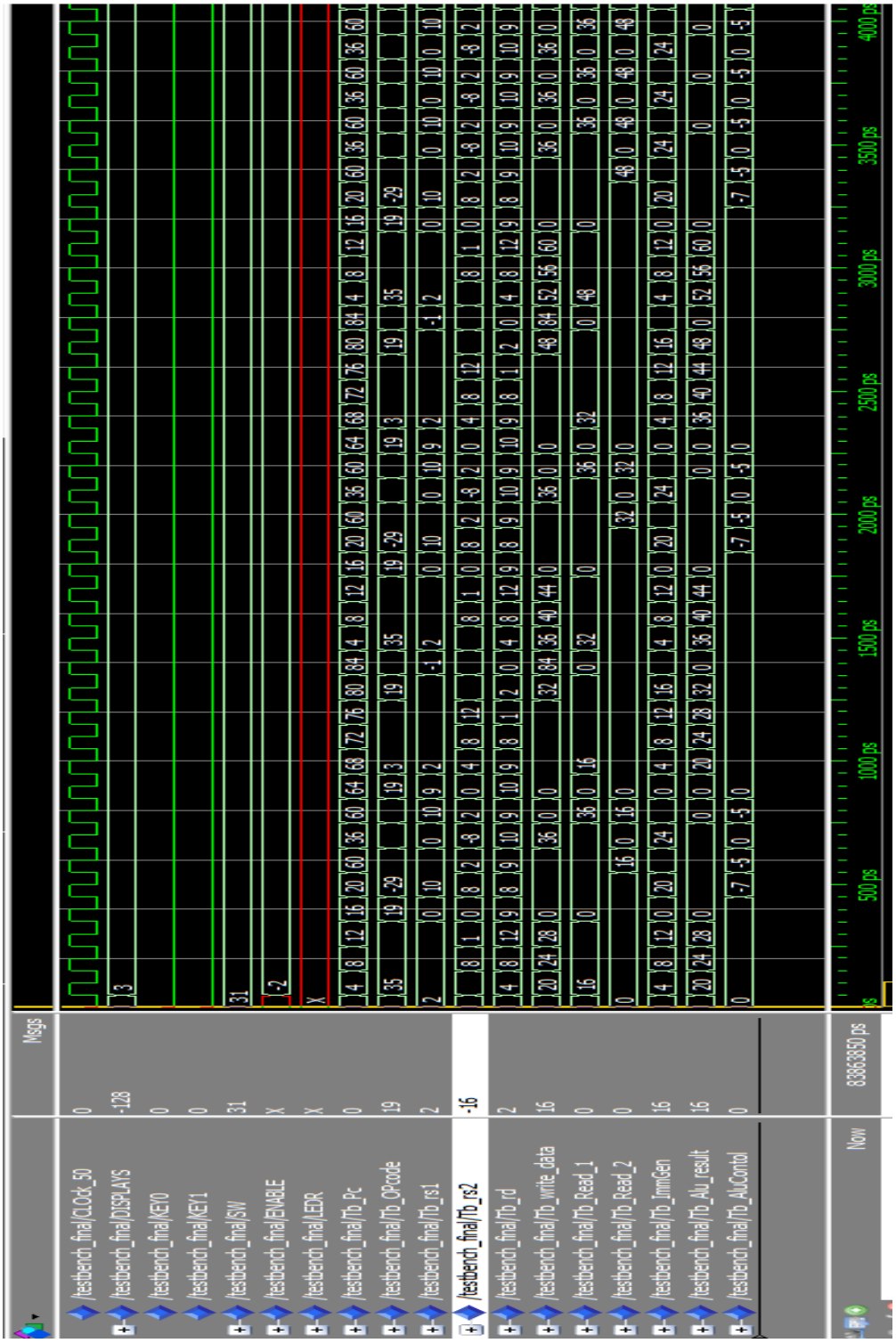
APÊNDICE C

Programa teste 2: Uma rotina em que soma os valores em uma árvore binária, usando uma travessia em ordem.

addi x2 x2 16 #aloca quadro de pilha	“00000001","00000001","00000001","00010011",
sw x18 4(x2) #preserva x2	"00000001","00100001","00100010","00100011",
sw x8 8(x2) #preserva x18	"00000000","10000001","00100100","00100011",
sw x1 12(x2) #preserva x8	"00000000","00010001","00100110","00100011",
addi x9 x0 0 #soma 0	"00000000","00000000","00000100","10010011",
beq x10 x8 L1 #pula o laço se o nodo ==0	"00000010","10000101","00000100","01100011",
addi x8 x10 0 #x8 = node	"00000000","00000101","00000100","00010011",
L3: lw x10 0(x8) #x10 = nodo > esquerda	"00000000","00000100","00100101","00000011",
jal x10 L1 #resulta em x10	"00000000","00000100","00100101","00000011",
L5:lw x15 8(x8) #x15 = nodo >valor	"00000001","10000000","00000101","01101111",
lw x8 4(x8) # nodo = nodo>direita	"00000000","10000100","00100111","10000011",
add x9 x10 x9 # soma+=x10	"00000000","01000100","00100100","00000011",
add x9 x9 x15 #soma+=x15	"00000000","10010101","00000100","10110011",
bne x8 x0 L3 # loop se nodo != 0	"00000000","11110100","10000100","10110011",
L1:addi x10 x9 0 #retorna a soma em a0	"11111110","00000100","00010010","11100011",
blt x10,x2,L5	"11111110","00100101","01000100","11100011",
lw x9 4(x2) #restaura valor de x9	"00000000","00000100","10000101","00010011",
lw x8 8(x2) #restaura o valor de x8	"00000000","01000001","00100100","10000011",
lw x1 12(x2) #restaura o valor de x1	"00000000","10000001","00100100","00000011",
addi x2 x2 16 # desloca o quadro de pilha	"00000000","11000001","00100000","10000011",
jalr x0 x1 0 # retorno	"00000001","00000001","00000001","00010011",

APÊNDICE D

Simulação RTL do programa 2 no ModelSim



APÊNDICE E

Programa 3 : Cálculo do Delta formula de Bháskara para teste da multiplicação

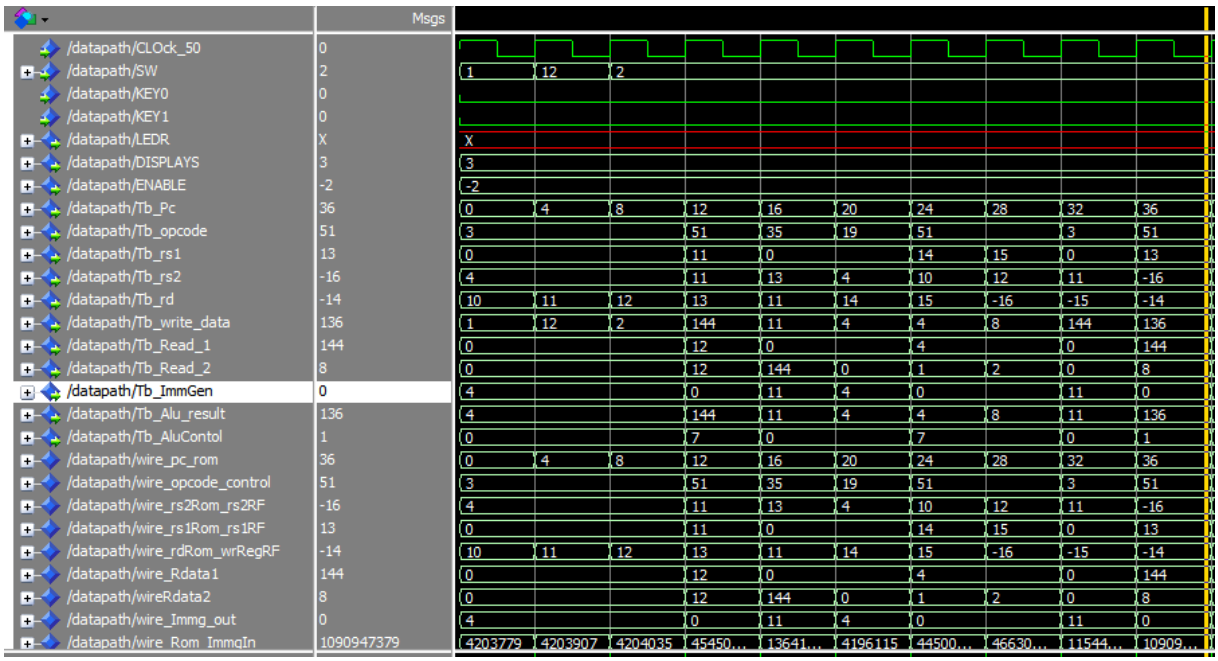
```
lw x10, 4(x0)
lw x11 ,4(x0)
lw x12 ,4(x0)
mul x13, x11,x11
sw x13 ,11(x0)
addi x14 x0 4
mul x15,x14,x10
mul x16,x15,x12
lw x17, 11(x0)
sub x18,x13,x16
```

valores de entrada

a=1;b=12;c=2

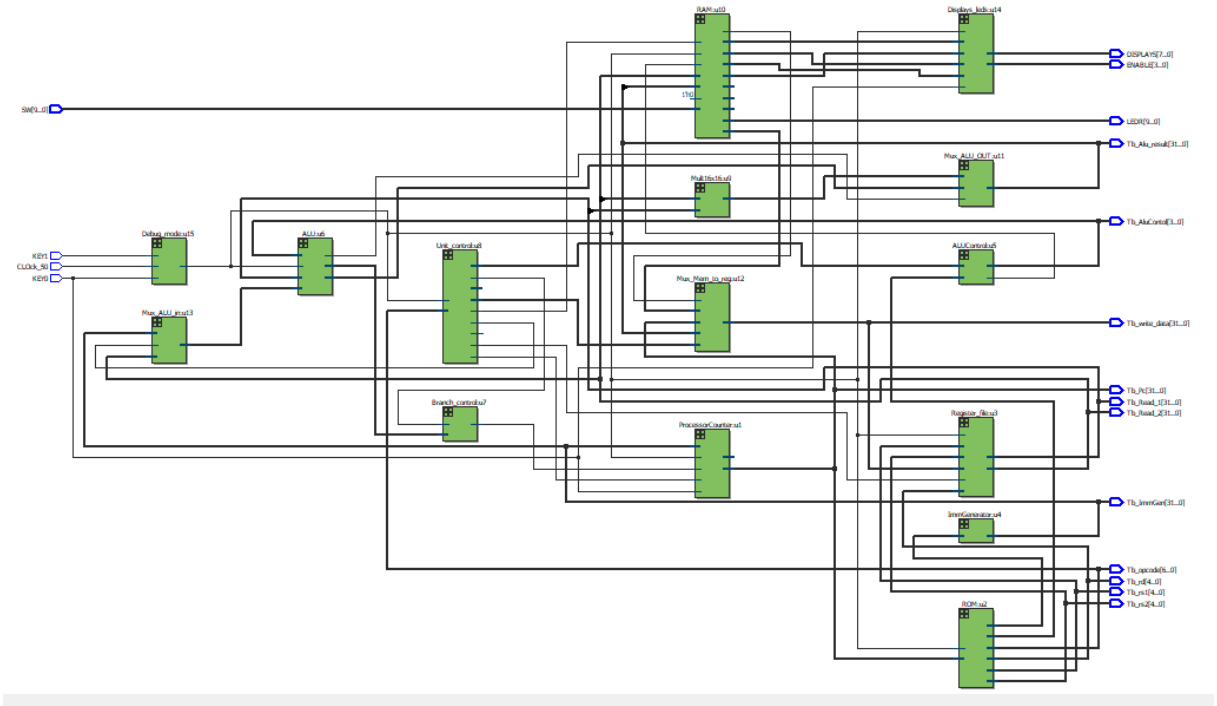
delta = 136 alu result

```
"00000000","01000000","00100101","00000011",
"00000000","01000000","00100101","10000011",
"00000000","01000000","00100110","00000011",
"00000010","10110101","10000110","10110011",
"00000000","11010000","00100101","10100011",
"00000000","01000000","00000111","00010011",
"00000010","10100111","00000111","10110011",
"00000010","11000111","10001000","00110011",
"00000000","10110000","00101000","10000011",
"01000001","00000110","10001001","00110011",
```



APÊNDICE F

RTL VIEW



ANEXO A

Tabela de codificação de instrução RV32I

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK