UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Kevin Paula Morais

DESCRIÇÃO EM VHDL DE UM PROCESSADOR RISC-V RV32EC

Santa Maria, RS
2018

Kevin Paula Morais

DESCRIÇÃO EM VHDL DE UM PROCESSADOR RISC-V RV32EC

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica, Área de Concentração em CNPq, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Engenheiro Eletricista em Engenharia Elétrica**.

ORIENTADOR: Prof. Giovani Baratto

Santa Maria, RS
2018

Kevin Paula Morais

DESCRIÇÃO EM VHDL DE UM PROCESSADOR RISC-V RV32EC

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica, Área de Concentração em CNPq, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Engenheiro Eletricista em Engenharia Elétrica**.

Aprovado em 14 de dezembro de 2018:

Giovani Baratto, Dr. (UFSM)
(Presidente/Orientador)

Mateus Beck Rutzig, Dr. (UFSM)

Michael Guilherme Jordan, Eng. (UFSM)

Santa Maria, RS
2018

AGRADECIMENTOS

Gostaria de agradecer aos meus pais, Marion dos Santos Morais e Lisete Cilene Paula Morais, que me proporcionaram suporte ao longo de toda a minha educação, desde o ensino fundamental, até o superior, e continuarão. A minha irmã, Tatiele Paula Morais Padilha, que sempre foi minha companheira em todos os assuntos, me auxiliando para um melhor caminho.

Gostaria de agradecer ao meu primo Maicon Morais Campos, com o qual tive a oportunidade de morar junto durante metade da graduação, e a minha prima Jéssica Paula Gonçalves, que me deu a felicidade de estar ao meu lado durante uma etapa do curso.

Gostaria de agradecer as minhas amigas Tamara Alves e Fernanda Martins, por sempre estarem comigo todos os dias, para conversas de assuntos diversos que me distraiam das complicações do curso e me divertiram tanto.

Gostaria de agradecer aos meus amigos Jean Rossini, Marcel Dall Pai e Victor Refosco, que conviveram comigo diariamente durante esse desafio que foi a graduação, e também durante momentos de festas e comemorações.

Gostaria de agradecer a todos os professores que se dedicaram a compartilhar de sua experiência para contribuir ao meu desenvolvimento, em especial ao Prof. Giovani Baratto pela orientação no desenvolvimento deste trabalho.

RESUMO

DESCRIÇÃO EM VHDL DE UM PROCESSADOR RISC-V RV32EC

AUTOR: Kevin Paula Morais
ORIENTADOR: Giovani Baratto

O projeto de processadores tem como ponto inicial suas especificações, considerando-se a aplicação, o tamanho e tipo dos dados a serem processados, e a arquitetura de conjunto de instruções (ISA) executado pelo hardware. Em muitas ISAs o licenciamento não é livre, dificultando as pesquisas para fins educacionais e acadêmicos. Neste trabalho utiliza-se da base E e extensões C da ISA RISC-V, uma ISA gratuita e de código aberto, no projeto de um processador de 32 bits, descrito em VHDL. A síntese e implementação do projeto foram realizadas no software Xilinx ISE Design Suite 14.7, na FPGA Spartan-6 XC6LX16-CS324, disponível na placa de desenvolvimento Nexys 3. A base E utiliza dados inteiros de 32 bits, com 16 registradores de uso geral, enquanto a extensão C proporciona instruções compressas de 16 bits. Estudou-se as instruções a serem implementadas, identificando-se as unidades necessárias para o desenvolvimento de um processador para dois núcleos diferentes, um multicíclico e outro pipeline de 5 estágios. O processador com núcleo multicíclico executa somente instruções da extensão C, RVC, enquanto que o pipeline executa o conjunto completo, RV32EC. Projetou-se os componentes necessários para a execução prática do núcleo, sendo eles a memória de programa e dados, e dois dispositivos E/S. Ambos os projetos foram implementados na FPGA com uma velocidade mínima de 100 MHz. Na implementação do multicíclico ficou claro o desperdício de se utilizar a extensão C sozinha, visto que ela ganha espaço em software, e não hardware. Enquanto que no pipeline percebeu-se o efeito da latência da memória no núcleo, que acabou por ignorar caminhos específicos do hardware.

Palavras-chave: Conjunto de Instruções. RISC-V. Processador. Pipeline. FPGA.

ABSTRACT

DESCRIPTION IN VHDL OF A RISC-V RV32EC PROCESSOR

AUTHOR: Kevin Paula Morais
ADVISOR: Giovani Baratto

The design of processors has as its starting point its specifications, considering the application, the size and type of data to be processed, and the instruction set architecture (ISA) executed by the hardware. In many ISAs, licensing is not free, making it difficult for research for educational and academic purposes. In this work we use the base E and extensions C of ISA RISC-V, a free and open source ISA, in the design of a 32-bit processor described in VHDL. The synthesis and implementation of the design were performed in the Xilinx ISE Design Suite 14.7 software, on the Spartan-6 XC6LX16-CS324 FPGA, available on the Nexys 3 development board. Base E uses 32-bit integer data, with 16 general purpose registers, while extension C provides 16-bit compress instructions. Was studied the instructions to be implemented, identifying the units necessary for the development of a processor with two different core, a multicycle and a 5-stage pipeline. The multicycle core processor executes only C extension instructions, RVC, while the pipeline runs the complete set, RV32EC. The necessary components for the practical execution of the core were designed, being the program and data memory, and two I/O devices. Both designs were implemented in the FPGA with a minimum speed of 100 MHz. In the implementation of the multicycle it was clear the waste of using the C extension alone, since it gains space in software, not hardware. While in the pipeline is was noticed the effect of memory latency upon the core, which ended ignoring specifics paths of the hardware.

Keywords: Instruction Set. RISC-V. Processor. Pipeline. FPGA.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABI	Application Binary Interface
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machine
$ASIC$	Application Specific Integrated Circuits
BHT	Branch History Table
$BRAM$	Block RAM
BTB	Branch Target Buffer
$CISC$	Complex Instruction Set Computer
CPU	Central Processing Unit
DSP	Digital Signal Processor
E/S	Entrada-Saída
$FPGA$	Field Programmable Gate Array
GND	Ground
IoT	Internet of Things
ISA	Instruction Set Architecture
LSB	Least Significant Bit
LUT	Look Up Table
MMU	Memory Management Unit
MSB	Most Significant Bit
PC	Program Counter
PCM	Phase-Change Memory
PLD	Programmable Logic Device
RAM	Random Acess Memory
RAS	Return Address Stack
$RISC$	Reduced Instruction Set Computer
ROM	Read Only Memory
SP	Stack Pointer

SO	Sistema Operacional
SoC	System-on-Chip
$UART$	Universal Asynchrhonous Receiver-Transmitter
VGA	Video Graphics Array
$VHDL$	Very High Speed Integrated Circuits Hardware Description Language

SUMÁRIO

1 INTRODUÇÃO

Em um processador, cada operação de um programa executado por ele é denominada de instrução, sendo ela pertencente a um conjunto, que exerce grande influência no projeto de hardware final do processador. De acordo com o conjunto de instruções (ISA) se obtém não somente circuitos de tamanho diferente, como também de aplicações especificas. Como por exemplo os processadores da Intel x86 de uso geral, ou ARM focados em sistemas embarcados, ou ainda mais concentrados em um uso, como DSPs. Os conjuntos de instruções podem ser divididos em dois tipos, CISC (complex instruction set computer) e RISC (reduced instruction set computer).

Uma das principais características presentes no conjunto CISC é o rico conjunto de instruções, enfase em operações orientadas a memória, e instruções de tamanho variado. Outro ponto importante é o tempo de execuções das instruções, que pode variar em diversos ciclos de clock, sendo também necessário inúmeras instruções RISC para formar uma CISC. Como exemplo de implementação do conjunto CISC na atualidade, pode-se utilizar a arquitetura x86 da Intel. Já o conjunto RISC opta por instruções mais simples, capazes de executar uma micro-operação apenas, sendo o tamanho da instrução fixo. Suas operações são voltadas para o uso de registradores, e possuem acessos a memória geralmente apenas por instruções simples de LOAD/STORE (GEORGE, 1990). Um exemplo do uso de núcleos RISC é a arquitetura utilizada nos projetos da ARM.

Atualmente se possui diversas ISAs possíveis para a implementação de hardware do processador, como a do MIPS utilizado didaticamente por (PATTERSON; HENNESSY, 2004), em que se aborda a implementação de dois tipos de núcleos diferentes, multicíclico e pipeline. O núcleo multicíclico tem como característica a execução de uma instrução em diversos ciclos de relógio, de acordo com os componentes que a mesma necessita, a fim de variar o tempo de execução das instruções de acordo com a necessidade. A técnica de pipeline, de forma semelhante a multicíclica utiliza diversos ciclos para execução de instruções, divididas por estágios. Porém no caso do pipeline, conforme certos componentes do núcleo concluem sua finalidade, os mesmos já são reutilizados no ciclo seguinte para a próxima instrução, possibilitando a execução em paralelo de instruções.

A grande maioria da ISAs não podem ser utilizadas livremente devido a questões de licenciamento, se tornando um obstáculo no avanço de pesquisas educacionais e/ou científicas, ou até mesmo com propósitos industriais. Outro fator importante é o nível de complexidade de alguns conjuntos de instruções, podendo não se tornar viável a sua implementação. O RISC-V surge como uma proposta de ISA para livre uso, proporcionando liberdade de pesquisa sobre a mesma no projeto de processadores de arquitetura RISC, sendo o seu projeto iniciado na Universidade da Califórnia, em Berkeley no ano de 2010. A arquitetura consiste em diversas extensões, que podem ser implementadas ou não, por

exemplo, instruções de multiplicação e divisão fazem parte de um conjunto de extensão especifico do RISC-V. Assim, basta não implementá-la se não desejarmos essa finalidade no hardware. Essa característica garante flexibilidade para o uso que se deseja atribuir ao núcleo a ser projetado.

Este trabalho consiste no uso da ISA do RISC-V, base E e extensão C mais precisamente, para projeto de um núcleo multicíclico e um com a técnica de pipeline. Projetando-se também de outros elementos essenciais, como memória e dispositivos periféricos, descrevendo então os componentes na linguagem de descrição de hardware VHDL, e implementado o hardware em um dispositivo FPGA.

1.1 OBJETIVO GERAL

O projeto consiste na descrição em VHDL de um processador multicíclico e um pipeline, que executam um subconjunto das instruções RV32EC, da ISA RISC-V, realizando a síntese e implementação na FPGA pertencente a família Spartan-6.

1.2 OBJETIVOS ESPECÍFICOS

O projeto tem como objetivos específicos:

• Estudo das instruções da base E e extensão C do RISC-V de (WATERMAN; ASANOVIć, 2017a), determinando as que serão implementadas;

• Projeto e descrição em VHDL de um caminho de dados multicíclico, capaz de executar as instruções da extensão C;

• Estudo sobre a técnica de pipeline, e de fatores relevantes a sua performance, como dependências de dados e controle;

• Projeto e descrição de um caminho de dados com a técnica de pipeline, capaz de executar as instruções da base E e extensão C;

• Projeto e descrição em VHDL dos dispositivos de memória utilizados para armazenamento de programa e dados;

• Projeto e descrição em VHDL de dispositivos periféricos para comunicação com o núcleo implementado;

• Estudo dos recursos disponíveis na placa de desenvolvimento Nexys 3, e do dispositivo FPGA encontrado na mesma para uso, Spartan-6;

• Implementação e teste de ambos os núcleos, multicíclico e pipeline, obtendo o uso de recursos lógicos da FPGA Spartan-6.

1.3 MOTIVAÇÃO

O uso de hardwares com tamanho reduzido em sistemas embarcados leva em consideração tanto o núcleo, quanto a memória, quando se fala em processadores. Esses aspectos visam a redução do consumo de energia, a fim de que em sistemas portáteis os mesmos tenham a maior duração possível em funcionamento, assim como também uma menor área do CI. A recente demanda por dispositivos voltados a IoT (Internet of Things) requer justamente dispositivos capaz de cumprir com essas necessidades, para se obter dispositivos com portabilidade.

Diferentes ISAs já procuraram utilizar instruções de tamanho reduzido, da mesma forma que o RISC-V propõe, que tem por objetivo reduzir o tamanho do código a ser executado pelo processador. Essa abordagem que reduz o tamanho do software reflete então na possibilidade de se utilizar memórias menores (ou necessidade), sendo esse justamente um dos componentes de maior área dentro dos sistemas. Além da memória principal (que armazena o programa a ser executado) ser responsável por uma parcela considerável do hardware, outro fator é o banco de registradores, elemento interno do núcleo. Este que é basicamente um elemento de memória, sendo no RISC-V de 32x32 bits, é responsável por uma quantia considerável de área do núcleo. Na base E do RISC-V modifica-se o banco para 32x16 bits, reduzindo-se pela metade seu tamanho, assim poupando área do núcleo, redução essa visando implementações dos menores microcontroladores possíveis de 32 bits, da ISA RISC-V.

Atualmente é comum o uso da técnica de pipeline nos núcleos, aumentando o desempenho dos processadores, e em casos de máquinas de uso geral, se tem ainda a implementação de arquiteturas superescalar. Como o uso do superescalar reflete em um aumento grande no hardware, o mesmo não é normalmente encontrado em sistemas embarcados, sendo geralmente utilizado apenas pipelines com execução em ordem e números de estágios variados (mas não muito profundos).

1.4 ESPECIFICAÇÕES DE PROJETO

Algumas definições de projeto devem ser atendidas conforme o conjunto de instruções a ser implementado do RISC-V (WATERMAN; ASANOVIć, 2017a), sendo o RV32EC, que se caracteriza por:

• Instruções formadas por palavras de 32 para a base E, e compressas de 16 bits para a extensão C;

• Dados com comprimento de 32 bits;

• Um total de 16 registradores de uso geral, de acordo com o uso da base E;

• Manipulação de dados unicamente do tipo inteiro.

A respeito do núcleo com a técnica de pipeline:

• O núcleo pipeline será projetado com base no modelo de 5 estágios apresentado por Patterson e Hennessy (2004);

• Tanto a emissão quanto a finalização das instruções serão realizadas em ordem;

A placa de desenvolvimento utilizada no projeto é a Nexys 3, que conta com:

• FPGA Spartan-6 XC6SLX16 CSG324C;

• Dispositivo de comunicação serial UART, conectado a uma porta micro USB;

• Cristal oscilador de 100 MHz;

• IDE para edição, síntese e implementação de projetos HDL, Xilinx ISE Design Suit 14.7.

Para a escrita de programas a serem executados no núcleo, se tem disponível as ferramentas GNU do RISC-V, contando com compiladores para as linguagens C e ASM. No momento de desenvolvimento do projeto, o suporte a linguagem C ainda não é pleno para a base E do RISC-V utilizada, visto que a mesma ainda está em desenvolvimento.

1.5 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta as principais características da ISA utilizada no trabalho, o RISC-V, dando foco ao conjunto de instruções do mesmo a ser implementado, RV32EC. Realiza-se também uma revisão teórica sobre projetos de processadores, desde o básico com conceitos de computadores genéricos, até estruturas com a técnica de pipeline. Por fim apresenta alguns trabalhos sobre a ISA do RISC-V.

O Capítulo 3 faz a apresentação da metodologia utilizada no decorrer do projeto, definindo-se as especificações iniciais, com base na ISA utilizada, que influenciaram no projeto do hardware.

O Capítulo 4 demonstra os resultados obtidos por meio da síntese e implementação dos componentes projetados, através de dados de consumo do dispositivo FPGA utilizado. Contém também simulações a fim de comprovar o funcionamento dos dispositivos, e de-

monstrar suas características de execução.

O Capítulo 5 apresenta a conclusão do projeto de acordo com os resultados obtidos, com propostas de trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo se realiza a revisão de conceitos importantes para a compreensão do projeto de um processador. A Seção 2.1 possuí uma breve descrição das arquiteturas RISC e CISC, seguida na Seção 2.2 pela apresentação da arquitetura de conjunto de instruções utilizada no projeto, o RISC-V. Na Seção 2.3 se encontra uma visão sobre a arquitetura e organização de computadores. Na Seção 2.4 é revisado o funcionamento de uma FPGA, dispositivo essencial para este trabalho, apresentando a placa de desenvolvimento utilizada no projeto. Por fim, na Seção 2.5 se tem alguns trabalhos já realizados sobre a ISA do RISC-V.

2.1 CONJUNTOS RISC E CISC

As operações de um processador são determinadas pelas suas instruções executadas, que ao serem agrupadas de forma a se estabelecer um padrão, são denominadas Conjunto de Instruções, se tem dois tipos de Arquiteturas de Conjunto de Instruções (ISA - Instruction Set Architecture), RISC e CISC. O RISC (Reduced Instruction Set Architecture) possuí como características base a execução de uma instrução por ciclo, formatos e modos de endereçamento simples (STALLINGS, 2012, p. 547), assim possibilitando o projeto de um hardware com complexidade reduzida. As operações em grande parte são realizadas entre registradores, e o acesso a memória se dá apenas por meio de instruções load e store, a implementação do controle é simplificada devido as instruções possuírem o tamanho fixo, e também a posição de seus campos de decodificação não variarem de posição. Em uma abordagem RISC também é relativamente simples de se implementar a técnica de pipeline.

Atualmente uma referência em arquiteturas RISC são os processadores ARM, populares em sistemas embarcados, já para computadores de uso geral encontramos a família x86 da Intel utilizando uma arquitetura de conjunto de instruções CISC. A ISA é caracterizada pela execução das instruções em múltiplos ciclos de relógio, sendo o tamanho delas também não fixo. A sua complexidade dificulta o projeto do pipeline do hardware em comparação com o RISC. Atualmente é possível encontrar implementações de processadores que realizam uma abordagem CISC quanto a ISA, porém RISC na arquitetura do hardware, a fim de combinar vantagens oferecidas por ambas as arquiteturas (STALLINGS, 2012). Atualmente o conjunto CISC implementado na arquitetura 80x86 da Intel utiliza da tradução por hardware de instruções para uma aproximação do tipo RISC, possibilitando o uso do ganho de performance de processadores RISC, sendo nomeado pela Intel de "micro-operações"(PATTERSON, 2018).

(BLEM; MENON; SANKARALINGAM, 2013) realiza uma comparação de eficiência da arquitetura RISC e CISC em termos de consumo de energia, utilizando como exemplo os microprocessadores da ARM Cortex-A8 e Cortex-A9 como modelo RISC, e para o CISC o modelo da Intel Atom e Sandybridge i7. As medições envolveram a carga computacional considerando dispositivos móveis, desktop, e servidores. Os resultados obtidos demonstraram que a diferença de ambos, ARM e x86, possuem questões de projeto voltadas a performance em diferentes áreas de análise, e são relativamente irrelevantes quando comparados em termos de consumo de energia. Essa análise considerou claro a questão de que apesar de o ARM estar predominantemente em sistemas embarcados e o x86 em em desktops potentes, os mesmos estão tentando expandir sua área, ARM para computadores de alta performance, e o x86 em direção a dispositivos móveis de baixo consumo de energia.

2.2 RISC-V

Surgiu inicialmente com o objetivo de se obter um conjunto de instruções mais adequado para fins de educação e pesquisa na área de projeto de processadores, sendo o seu licenciamento livre para aplicações tanto acadêmicas quanto industriais. O RISC-V em sua base utiliza instruções com comprimento de 32 bits, possui quatro versões bases na sua ISA, todas com dados do tipo inteiro, diferenciando-se no tamanho dos dados, sendo duas de 32 bits, e as outras de 64 e 128 bits, as bases de 32 bits diferem no número de registradores para uso geral disponíveis. A fim de se obter uma maior variedade de implementações do RISC-V, extensões da ISA base foram projetados de forma a expandir sua arquitetura, sendo referenciadas por letras, por exemplo a extensão M que acrescenta instruções de multiplicação e divisão ao conjunto. Para uma versão base de 32 bits com o acréscimo da extensão M, se utiliza a nomenclatura RV32IM, o mesmo se aplica para a adição de outras extensões. A Tabela 2.1 apresenta todas as bases e extensões padrões do RISC-V. Ao uso da base I com as extensões MAFD, se é utilizada a letra "G"na nomenclatura, RV32IG, como uma ISA padrão de propósitos gerais (WATERMAN; ASANOVIć, 2017a, p. 121). Apesar da variedade de extensões do RISC-V, nem todas são compatíveis entre si.

Todas as instruções base do RISC-V possuem os dois bits menos significativos fixos em nível ’1’, identificando assim o tamanho da instrução como 32 bits, a alteração desses bits proporcionam outros comprimentos para as instruções, todos em parcelas de 16-bits, conforme a codificação da Figura 2.1. Por padrão a ISA base possui um sistema de memória little-endian, o que proporciona a possibilidade da verificação dos bits de codificação do comprimento da instrução com prioridade no estágio de busca da instrução.

A base do RISC-V conta com 32 registradores a disposição, referenciados pela letra

Tabela 2.1 – Tabela da ISA do RISC-V.

Base	Extensão
RV32I	M - Integer Multiplication and Division
RV32E	A - Atomic
RV64I	F - Single-Precision Floating-point
RV128I	D - Double-Precision Floating-point
	Q - Quad-Precision Floating-point
	L - Decimal Floating-point
	C - 16-bits Compressed Instructions
	B - Bit Manipulation
	J - Dynamic Languages
	T - Transactional Memory
	P - Packed-SIMD Extensions
	V - Vector Extensions
	N - User-Level Interrupts

Fonte: Adaptado de Waterman e Asanović (2017a, Preface p. i).

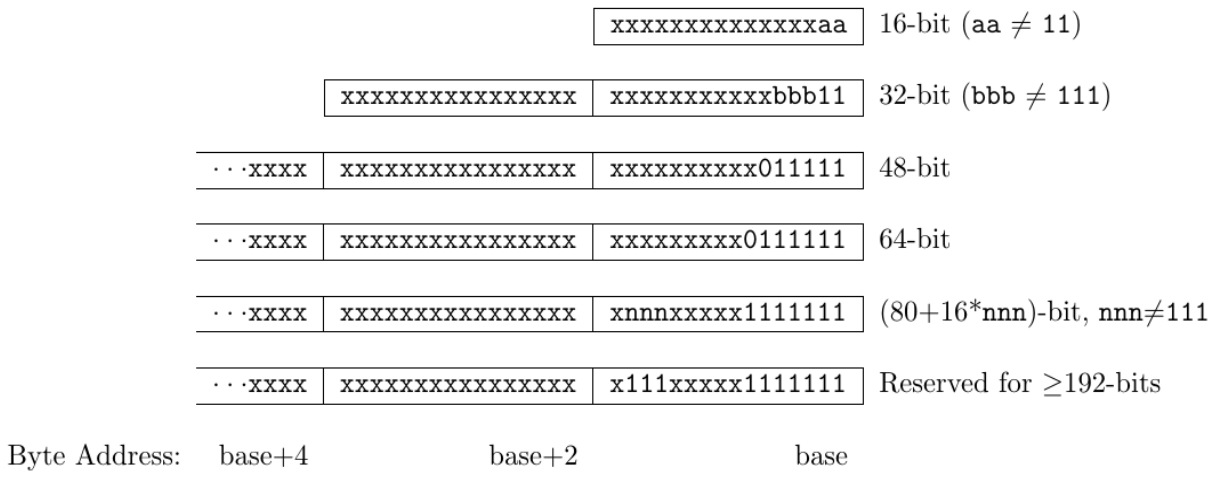
"x", x0 até x31, mais um registrador de uso especifico nomeado de Program Counter (PC). O x0 corresponde a constante 0, e o x1 por convenção se denomina como o link register para armazenamento de endereços de retorno, outras convenções da Application Binary Interface (ABI) podem ser consultadas em Waterman e Asanović (2017a, p. 109). Dentre as bases e extensões, duas são de grande utilidade no que se refere a obtenção de um núcleo reduzido, ideal para sistemas embarcados, a base E e a extensão C.

2.2.1 Base RV32E

Na base E limita-se os registradores de uso geral disponíveis de 32 para 16, logo só temos acesso aos referenciados de x0 até x15, tentativas de uso dos demais resultaria em uma exceção de instrução ilegal do RV32I. O seu conjunto de instruções corresponde ao mesmo do RV32I. A Figura 2.2 contém a organização dos registradores de ambas as bases RV32I e RV32E, as duas bases restantes apenas variam o tamanho dos dados do núcleo de 0 até 64 ou 128 bits, para RV64I e RV128I, respectivamente. Nenhuma extensão padrão do RISC-V utiliza os bits livres da instrução, que correspondem aos campos antes utilizados para especificar acesso aos registradores de x16 a x31, sendo deixados a critério para implementações de extensões não padrões. A base E é compatível com as extensões M, A e C, não sendo suportadas as extensões de ponto flutuante, pois se tornaria uma contradição com o objetivo de ganhar área com a redução de registradores gerais (WATERMAN; ASANOVIć, 2017a, p. 27-28).

O conjunto de instruções do RV32E (mesmo do RV32I) possui 4 formatos de instruções, R, I, S e B, e 2 subformatos, U e J, representados na Figura 2.3 em conjunto com

Figura 2.1 – Codificação do comprimento das instruções do RISC-V.



Fonte: Waterman e Asanović (2017a, p. 6).

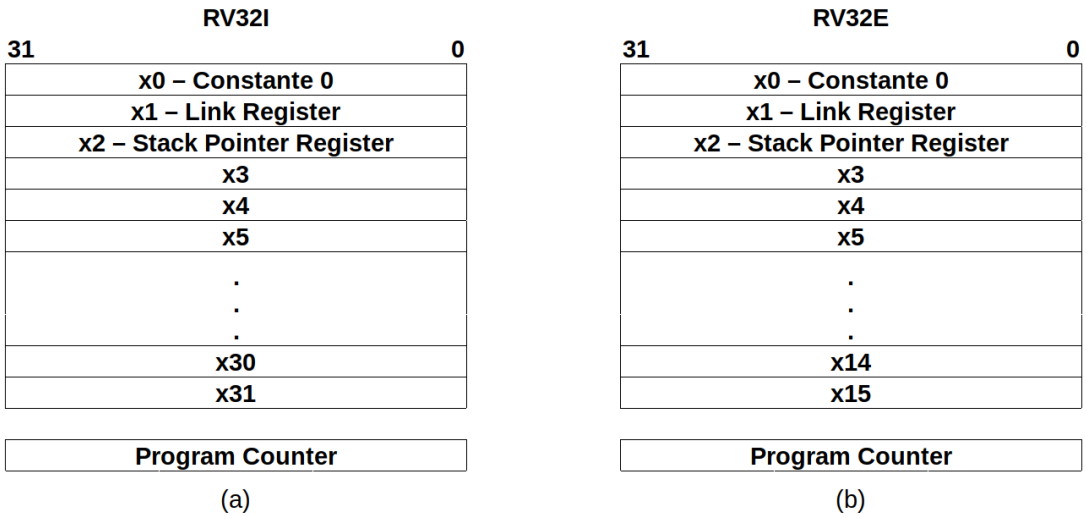
a lista de instruções. Os campos de leitura e escrita no banco de registradores, rs1, rs2 e rd, estão fixos na mesma posição, de forma a simplificar a decodificação, o mesmo vale para o campo opcode que é utilizado para decodificar a instrução atual, sendo necessário em algumas instruções o uso do campo funct3 para definir algumas operações internas do processador. Está explícito nos campos qual bit da instrução corresponde a qual bit do sinal imediato, que é embaralhado ao longo dos formatos das instruções. Isso é feito de forma com que seja necessário o menor número de multiplexadores possíveis na decodificação do imediato, o bit mais significativo por exemplo, está sempre posicionado no bit 31 da instrução, facilitando a implementação do hardware extensor de sinal. Outra vantagem é a dependência do campo imediato somente do campo opcode da instrução, logo sua decodificação pode ocorrer em paralelo com o restante dos sinais de controle do núcleo.

2.2.2 Extensão C

A extensão C proporciona um subconjunto de instruções da versão base de 32 bits, compressas para 16 bits, tipicamente 50-60% das instruções do RISC-V em um código podem ser substituídas pelas do RVC (denominação da extensão C aplicado ao RV32, RV64 ou RV128), resultando em uma redução de 25-30% do tamanho do código (WATERMAN; ASANOVIć, 2017a, p. 67).

A compressão é obtida por meio de definições como o valor do registrador destino (rd) sendo igual a um dos registradores fonte (rs1 ou rs2), ou então limitando o acesso dos registradores de x8 a x15, observando que essa limitação é compatível com a base E que tem 16 registradores de uso geral (x0 até x15). Quando se tem essa limitação de acesso

Figura 2.2 – (a) Registradores de uso geral RV32I; (b) Registradores de uso geral do RV32E.



Fonte: Adaptado de Waterman e Asanović (2017a, p. 10).

aos registradores, o campo da instrução é referenciado em conjunto com aspas simples, rs1', rs2' e rd'. Na Figura 2.4 se demonstra os oito formatos diferentes do subconjunto RVC, aonde se percebe que o campo imm (imediato) novamente está embaralhado, a fim de se obter um menor número de multiplexadores para sua decodificação. O bit mais significativo utilizado para estender o sinal do imediato está localizado sempre na posição 12 da instrução, facilitando assim a implementação do hardware extensor, o campo opcode também pode ser utilizado para a lógica de decodificação do imm, pois o sinal será estendido do bit mais significativo apenas quando opcode for 10, de resto (01 ou 10) temos que o imediato é estendido com 0s. A instrução é definida de acordo com o campo funct3 em conjunto com o opcode, porém quando os mesmos possuem mais de uma instrução para um mesmo valor, se utilizada os demais bits para decodificação.

Os registradores fonte, rs1 e rs2, estão fixos, com rd alternando entre eles, sendo que quando o registrador destino está especificado com todos os 5 bits, ele está na mesma posição em que as instruções da base. Essas considerações no projeto do RISC-V foram criadas para que cada instrução RVC possa ser expandida em uma equivalente da base de 32 bits durante a decodificação da instrução. Na Figura 2.5 demonstra-se todas as instruções RVC, aonde se percebe a codificação do tamanho da instrução nos dois bits menos significativos, que são iguais a 00, 01 ou 10, do contrário corresponde a uma instrução de comprimento maior que 16 bits. É importante notar que a extensão C não foi projetada para funcionar como uma stand-alone ISA.

Figura 2.3 – Formatos, subformatos e lista de instruções da base RV32E.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

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
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Fonte: Waterman e Asanović (2017a, p. 12).

Figura 2.4 – Formatos e campos das instruções da extensão C.

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

Fonte: Waterman e Asanović (2017a, p. 70).

2.3 ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES

Em uma visão de alto nível, um computador consiste em uma CPU (Unidade Central de Processamento), Memória e Dispositivos E/S, sendo todos conectados através de barramentos para transferência de dados, módulos, ou outra arquitetura e técnica de comunicação entre os componentes. Na Figura 2.6 se apresenta uma visão de alto nível de um computador. A CPU é responsável pelo controle da operação do computador em geral, processando dados, a memória principal é utilizada como armazenamento de dados, os dispositivos de entrada saída para comunicações externas do processador (CPU) (STALLINGS, 2012).

Um processador funciona basicamente realizando operações descritas por instruções de máquinas, que em conjunto formam uma ISA. Para isso, diversos componentes são fundamentais:

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

• Unidade Lógica Aritmética (ULA) - responsável pela execução de operações lógicas e aritméticas das instruções, como somas, substrações, comparações, lógicas OR e AND, entre outros tipos;

• Contador de Programa (Program Counter - PC) - registrador que aponta para a instrução da memória a ser buscada;

• Unidade de Decodificação - utilizada para decodificar os campos da instrução que correspondem a operação, controlando o núcleo;

• Registradores Temporários - registradores utilizados para armazenar dados como a instrução lida da memória, os endereços lidos do banco de registradores, ou então resultados da ULA.

Figura 2.5 – Lista de instruções RVC.

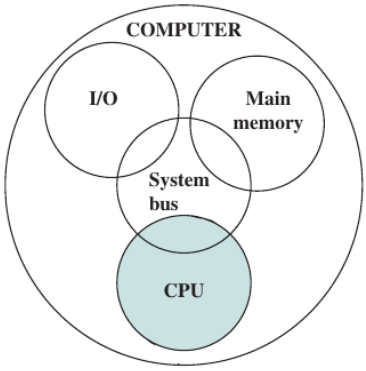
15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	*Illegal instruction*
000	nzuimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN *(RES, nzuimm=0)*
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.FLD *(RV32/64)*
001	uimm[5:4|8]	rs1′	uimm[7:6]	rd′	00	C.LQ *(RV128)*
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.FLW *(RV32)*
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.FSD *(RV32/64)*
101	uimm[5:4|8]	rs1′	uimm[7:6]	rs2′	00	C.SQ *(RV128)*
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.FSW *(RV32)*
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.SD *(RV64/128)*

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	0	0	0	01	C.NOP
000	nzimm[5]	rs1/rd≠0	nzimm[4:0]	01	C.ADDI *(HINT, nzimm=0)*
001	imm[11|4|9:8|10|6|7|3:1|5]			01	C.JAL *(RV32)*
001	imm[5]	rs1/rd≠0	imm[4:0]	01	C.ADDIW *(RV64/128; RES, rd=0)*
010	imm[5]	rd≠0	imm[4:0]	01	C.LI *(HINT, rd=0)*
011	nzimm[9]	2	nzimm[4|6|8:7|5]	01	C.ADDI16SP *(RES, nzimm=0)*
011	nzimm[17]	rd≠{0, 2}	nzimm[16:12]	01	C.LUI *(RES, nzimm=0; HINT, rd=0)*
100	nzuimm[5]	00 rs1′/rd′	nzuimm[4:0]	01	C.SRLI *(RV32 NSE, nzuimm[5]=1)*
100	0	00 rs1′/rd′	0	01	C.SRLI64 *(RV128; RV32/64 HINT)*
100	nzuimm[5]	01 rs1′/rd′	nzuimm[4:0]	01	C.SRAI *(RV32 NSE, nzuimm[5]=1)*
100	0	01 rs1′/rd′	0	01	C.SRAI64 *(RV128; RV32/64 HINT)*
100	imm[5]	10 rs1′/rd′	imm[4:0]	01	C.ANDI
100	0	11 rs1′/rd′	00 rs2′	01	C.SUB
100	0	11 rs1′/rd′	01 rs2′	01	C.XOR
100	0	11 rs1′/rd′	10 rs2′	01	C.OR
100	0	11 rs1′/rd′	11 rs2′	01	C.AND
100	1	11 rs1′/rd′	00 rs2′	01	C.SUBW *(RV64/128; RV32 RES)*
100	1	11 rs1′/rd′	01 rs2′	01	C.ADDW *(RV64/128; RV32 RES)*
100	1	11 —	10 —	01	*Reserved*
100	1	11 —	11 —	01	*Reserved*
101	imm[11|4|9:8|10|6|7|3:1|5]			01	C.J
110	imm[8|4:3]	rs1′	imm[7:6|2:1|5]	01	C.BEQZ
111	imm[8|4:3]	rs1′	imm[7:6|2:1|5]	01	C.BNEZ

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	C.SLLI *(HINT, rd=0; RV32 NSE, nzuimm[5]=1)*
000	0	rs1/rd≠0	0	10	C.SLLI64 *(RV128; RV32/64 HINT; HINT, rd=0)*
001	uimm[5]	rd	uimm[4:3|8:6]	10	C.FLDSP *(RV32/64)*
001	uimm[5]	rd≠0	uimm[4|9:6]	10	C.LQSP *(RV128; RES, rd=0)*
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	C.LWSP *(RES, rd=0)*
011	uimm[5]	rd	uimm[4:2|7:6]	10	C.FLWSP *(RV32)*
011	uimm[5]	rd≠0	uimm[4:3|8:6]	10	C.LDSP *(RV64/128; RES, rd=0)*
100	0	rs1≠0	0	10	C.JR *(RES, rs1=0)*
100	0	rd≠0	rs2≠0	10	C.MV *(HINT, rd=0)*
100	1	0	0	10	C.EBREAK
100	1	rs1≠0	0	10	C.JALR
100	1	rs1/rd≠0	rs2≠0	10	C.ADD *(HINT, rd=0)*
101	uimm[5:3|8:6]		rs2	10	C.FSDSP *(RV32/64)*
101	uimm[5:4|9:6]		rs2	10	C.SQSP *(RV128)*
110	uimm[5:2|7:6]		rs2	10	C.SWSP
111	uimm[5:2|7:6]		rs2	10	C.FSWSP *(RV32)*
111	uimm[5:3|8:6]		rs2	10	C.SDSP *(RV64/128)*

Fonte: Adaptado de Waterman e Asanović (2017a, p. 82 e 83).

Figura 2.6 – Visão de alto nível de um computador.



Fonte: Stallings (2012, p. 13).

Os campos das instruções utilizados pelo processador geralmente se definem nos utilizados para decodificação, o que possui de forma explicita os endereços de leitura do banco de registradores, sendo os registradores fonte (Register Source - rs), e também o de escrita do banco, registrador destino (Register Destiny - rd). Algumas instruções possuem campos denominados de imediato, que utilizados como um operando em determinadas operações, logo o uso da ULA pode ser sobre diferentes tipos de operando, como Registrador para Registrador (fonte banco de registradores), ou Registrador para Imediato, por exemplo. Os diferentes tipos de instruções podem ser definidos como:

• Transferência de Dados - realiza a transferência de dados, que pode ser internamente ao processador, ou externa com alguma memória de dados;

• Aritméticas e Lógicas - executam operações de soma, subtração, AND, OR, XOR, ou então deslocamentos aritméticos/lógicos, sobre dois operando;

• Conversão - utilizadas para converter tipos de dados, como binário para decimal;

• E/S - instruções dedicadas para uso de dispositivos de entrada/saída, sendo sua presença dependente da arquitetura do processador;

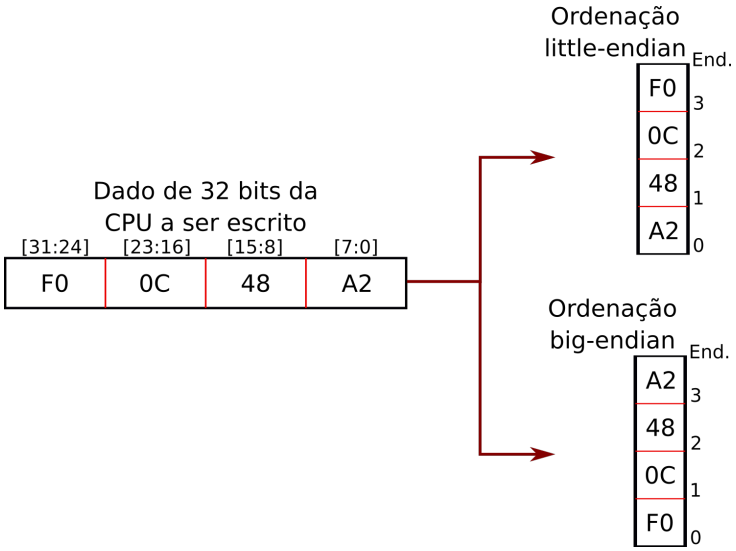
• Controle do Sistema - instruções de nível privilegiado do processador, tipicamente reservadas para uso do sistema operacional;

• Transferência de Controle - instruções que alteram o fluxo normal de execução de instrução para o qual o PC está indicando.

Em uma memória endereçada a byte, a ordem com que os dados são lidos/escritos podem ser de dois tipos, little-endian ou big-endian. Na ordenação little-endian temos que o byte menos significativo do dado é escrito/lido no endereço de menor valor a ser acessado, e de forma contrária, o big-endian armazena/lê o byte menos significativo no endereço de maior valor (STALLINGS, 2012, p. 447). O processo é descrito na Figura 2.7, onde se utiliza um dado a ser armazenado de valor hexadecimal F0 no byte mais significativo, e A2

no menos, o comprimento do dado é de 4 bytes.

Figura 2.7 – Ordenação dos bytes, little-endian e big-endian.



Fonte: Autor.

2.3.1 Processador Monociclo e Multicíclico

O núcleo da CPU, responsável pela execução da instrução, pode ter o seu comportamento descrito por meio de um caminho de dados, que consiste na representação do fluxo de dados dentre os diversos componentes necessários a sua execução. Patterson e Hennessy (2004) realizam uma abordagem didática sobre um caminho de dados do conjunto de instruções do processador MIPS, aonde se utiliza elementos como a ULA, banco de registradores, contador de programa, entre outros. Também se representa a memória a ser acessada para buscar instruções, e ler/escrever dados, apesar de o componente em si ser externo ao núcleo que executa a instrução. O hardware representado funciona de forma síncrona com o sinal de relógio, que dita a velocidade com que o mesmo trabalha, e nesse ponto é possível definir dois tipos de formas de execução das instruções, monocíclicas ou multicíclicas.

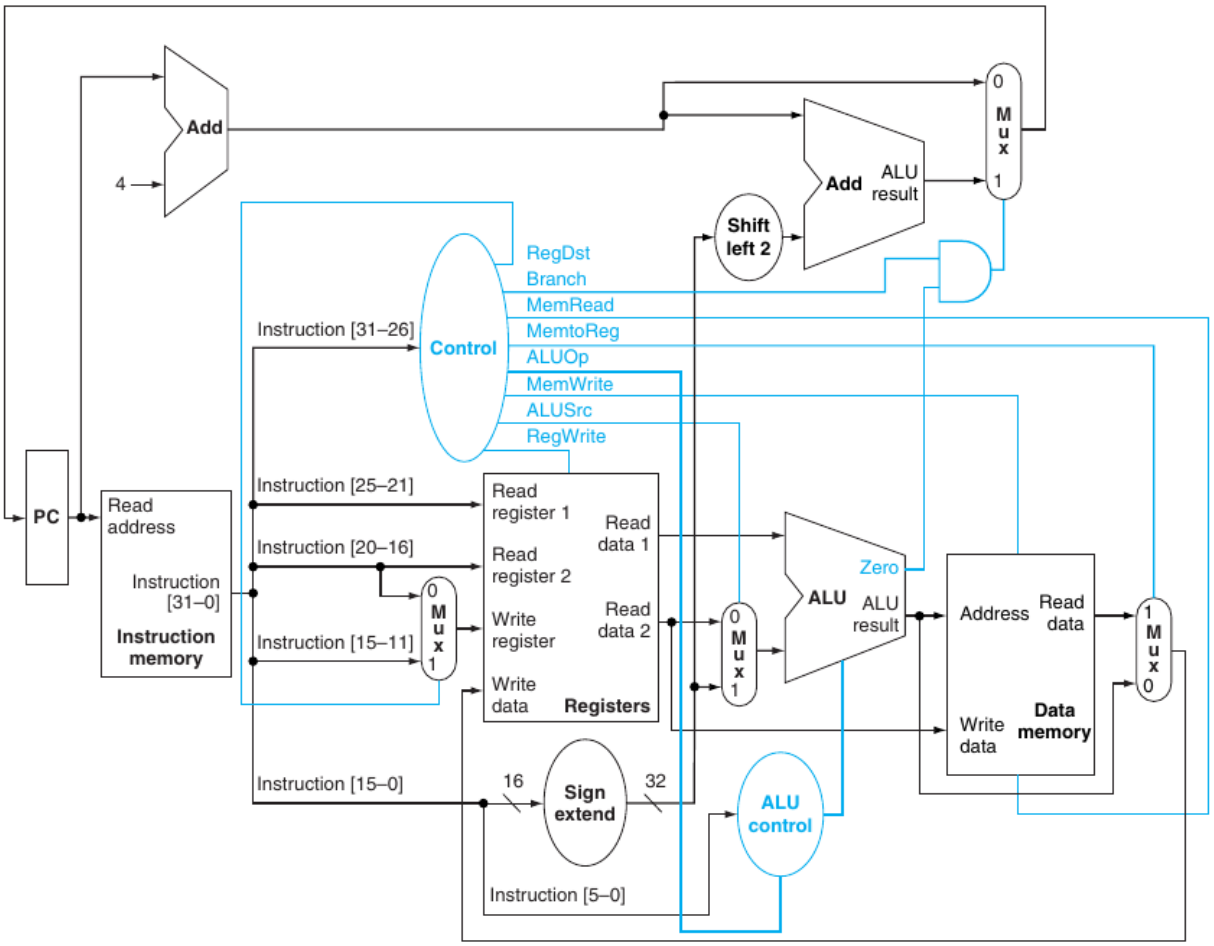
Para núcleos monocíclicos temos como exemplo de o caminho de dados da Figura 2.8, nele o fluxo de dados ocorre da esquerda para a direita, iniciando pelo PC (Program Counter), que aponta para o endereço da memória da instruções a ser acessado, em paralelo ocorre o incremento de PC para apontar a instrução que será buscada e executada no próximo ciclo de relógio. Após a obtenção da instrução, seus campos são separados para as diferentes unidades, os campos que determinam a operação para a unidade de controle (Control) que correspondem aos bits de 31 a 26, outros campos conectados ao banco de

registradores (Registers), indicando os endereços a serem lidos e o de escrita. E por fim os campos do sinal imediato, que são estendidos para 32 bits a fim de serem utilizados. O restante da operação do núcleo depende da decodificação da instrução, que consiste de um circuito combinacional, se for uma instrução do tipo aritmética/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 Load Word, se calcularia o 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. Maiores detalhes da execução das instruções ao longo do caminho de dados são descritas por Patterson e Hennessy (2004, cap. 4).

Como se percebe, numa abordagem monocíclica as instruções percorrem muitas vezes 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 relógio seja grande o suficiente para acomodar a mais lentas das instruções. Supondo que a uma instrução de soma e de leitura da memória, ADD e LW, sejam executadas em 10 ns e 14 ns, respectivamente, o sinal de relógio de ambas seria de 100 MHz e 71,4 MHz. A fim de que todas as instruções funcionem corretamente, o escolhido para o sistema seria de 71,4 MHz, assim fica evidente que apesar de se possuir uma vantagem de executar uma instrução por ciclo em um caminho monocíclico, o sinal de relógio tende a uma frequência baixa como desvantagem, o que nos leva ao multicíclico.

Em um caminho multicíclico, a instrução é dividida em mais de uma etapa de execução, por exemplo, a busca da instrução da memória consiste na primeira etapa, no próximo ciclo de relógio então se realiza a decodificação em paralelo com a leitura do banco de registradores, para o terceiro ciclo se tem a execução da operação na ALU, e por fim no quarto estágio o armazenamento do resultado no banco de registradores. Para que isso seja possível, é necessário modificar o esquema da Figura 2.8, na saída da memória de instruções se adiciona um registrador, que mantém a instrução buscada da memória até o fim de sua execução no quarto ciclo de relógio, outros registradores temporários são adicionados, um para guardar os dois dados lidos do banco de registradores, na saída da ALU, e para armazenar também dados buscados da memória em instruções de load. Os registradores servem como "barreiras temporais", que definem até onde vai a execução de determinado estágio da instrução. O controle passa a ser mais complexo, pois agora não basta um circuito combinacional que possui como entrada os bits de 31-26 da instrução, pois os sinais de controle devem alterar de valor a cada ciclo, conforme a etapa atual da instrução, para isso se utiliza uma máquina de estados finitos. Apesar dos registradores adicionados, e da adaptação do controle para uma máquina de estados, uma das características do caminho de dados multicíclico é a de reduzir o hardware, antes se possuía um total de três circuitos somadores de 32 bits, um para incrementar o valor de PC, outro dentro da ALU, e por fim um para calcular os endereços alvos de instruções de desvio

Figura 2.8 – Caminho de dados monocíclico, bits 31-26 da instrução correspondem a operação da instrução, 25-21 e 20-16 aos campos de leitura dos registradores 1 e 2 do banco de registradores, o endereço de escrita é definido por um multiplexador entre os campos 20-16 e 15-11, e os campos 15-0 carregam o sinal imediato a ter o bit mais significativo estendido.

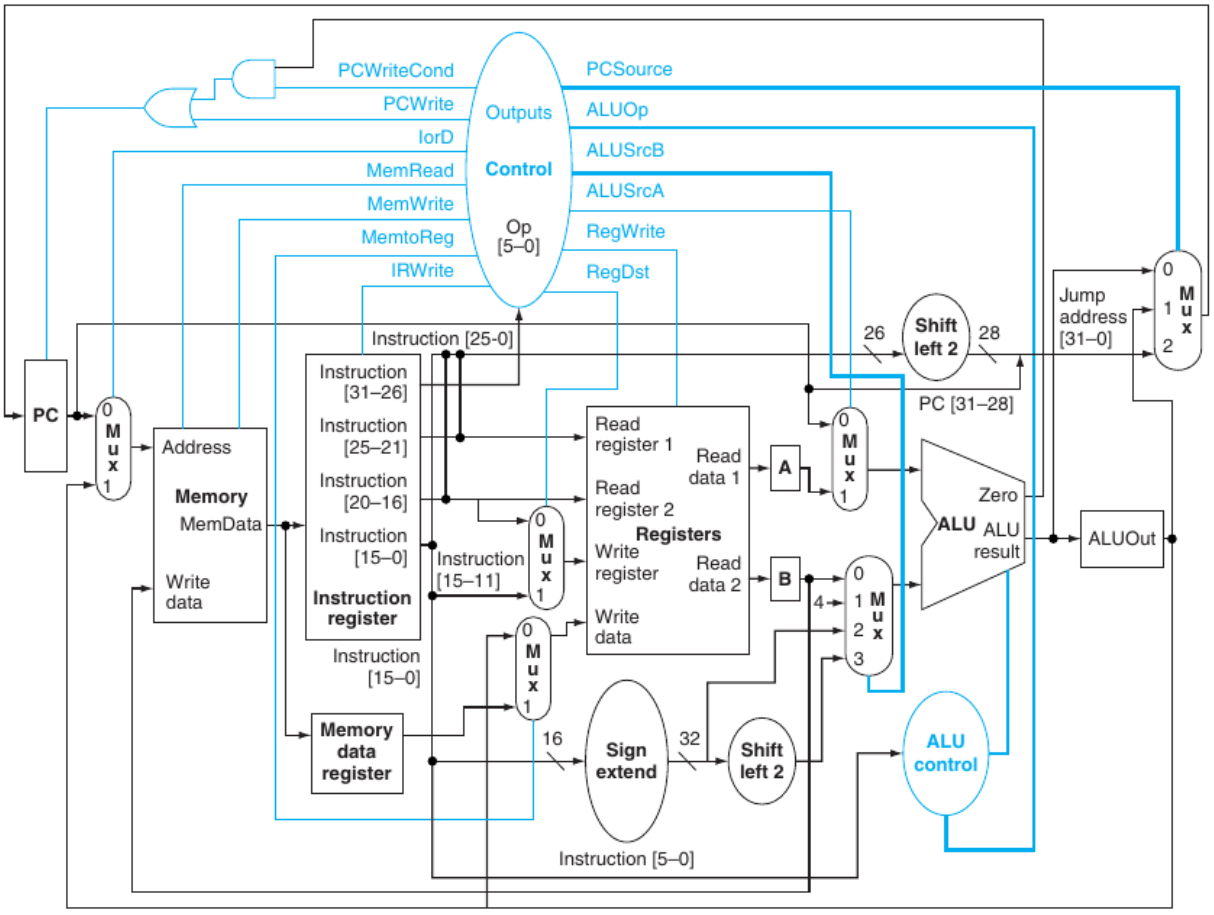


Fonte: Patterson e Hennessy (2004, p. 307).

(jump, branch). Porém, com a máquina de estados surge a possibilidade utilizar um único somador mais de uma vez, ao longo da execução total de uma única instrução, no primeiro ciclo por exemplo, enquanto se utiliza o valor de PC para acessar a memória, o mesmo é conectado a um multiplexador que escolhe um dos operando da ALU, realizando assim em paralelo o seu incremento, e descartando a necessidade de hardware dedicado a essa função. A mesma lógica se aplica para o somador antes utilizado para formar endereços de desvio, a Figura 2.9 demonstra o caminho de dados modificado.

O ganho da utilização de uma abordagem multicíclica está no fato de que ao dividir uma instrução em diversas etapas, o sinal de relógio deve então passar a satisfazer a etapa mais lenta, logo se a divisão for realizada de forma a manter aproximadamente igual o tempo de execução das etapas, o ganho de frequência do sinal de relógio é equivalente

Figura 2.9 – Caminho de dados multicíclico, registradores adicionados: Instruction register, Memory data register, A, B e ALUOut.



Fonte: Patterson e Hennessy (2004, p. 323).

ao número de etapas da instrução com maior número de ciclos. Apesar de se obter um aumento na frequência do sistema, se perde a vantagem do monocíclico, que se resume ao fato de possuir a execução de uma instrução por ciclo de relógio.

2.3.2 Pipeline

A técnica de pipeline consiste na execução de múltiplas instruções sobrepostas, sendo atualmente um ponto chave para o projeto de processadores mais rápidos (PATTERSON; HENNESSY, 2004, p. 370). Sua implementação consiste em dividir o caminho de dados em mais de um estágio de execução, como no multicíclico, porém o número de estágios será o mesmo para todas as instruções, independente de ela ser efetivamente concluída antes. São colocados registradores para armazenar as informações que devem ser mantidas das instruções em execução para o próximo estágio, nomeia-se esses de

registradores de pipeline, antes ao buscar uma instrução no primeiro estágio, o PC e a memória de instruções ficavam ociosos, esperando o termino do restante da instrução, que ainda levaria alguns ciclos de relógio, porém com a adição dos registradores de pipeline, que mantém o estado da instrução seguro a cada ciclo, é possível aproveitar os hardwares que estão inativos para dar andamento a execução das próximas instruções. Em uma situação com um núcleo que possuí um pipeline de 5 estágios, ao término de uma instrução no quinto, os quatro estágios anteriores já estão preenchidos cada um com uma instrução diferente, se obtendo assim a execução de uma instrução por sinal de relógio, e obtendo o ganho de frequência desejado com a divisão da execução do caminho de dados, que em teoria é um fator igual ao número de estágios de pipeline (quando perfeitamente distribuído). Patterson e Hennessy (2004) apresentam a técnica de pipeline implementada com 5 estágios, com as seguintes características:

• 1ºInstruction Fetch (IF): utiliza o endereço do contador de programa (PC) para acessar a memória de instruções e buscar a próxima a ser executada no pipeline, em paralelo se realiza o seu acréscimo para apontar para a seguinte da memória;

• 2ºInstruction Decode (ID): realiza a decodificação da instrução buscada e armazenada no registrador de pipeline IF/ID, em paralelo realiza a leitura do banco de registradores;

• 3ºExecute (EX): faz a execução lógica/aritmética da ALU, e também calcula em paralelo o endereço alvo para instruções de desvio condicional/incondicional;

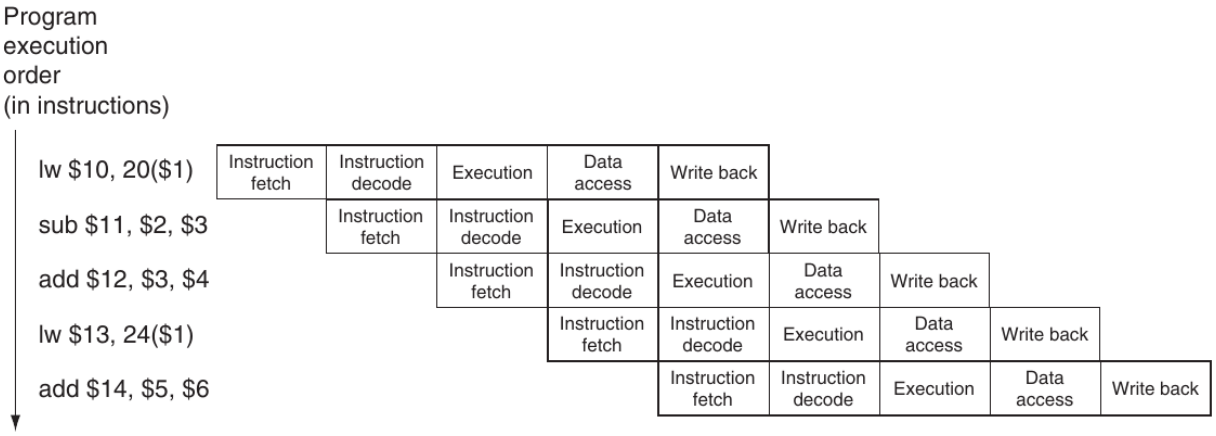
• 4ºMemory (MEM): acessa a memória para leitura/escrita de dados, sendo o endereço o resultado da operação da ALU armazenada no registrador de pipeline EX/MEM;

• 5ºWrite Back (WB): escreve o resultado da execução da instrução quando indicado pelo sinal de controle carregado ao longo dos registradores de pipeline;

A Figura 2.10 demonstra a execução de quatro instruções ao longo dos estágios do pipeline, aonde se possuí primeiro um atraso de quatro ciclos para preencher o pipeline caso ele ainda não esteja em execução, e a partir de então adquirir a finalização de uma instrução por ciclo de relógio.

O esquema do caminho de dados pode ser visualizado na Figura 2.11, onde fica evidente os registradores de pipeline entre os estágios, os sinais de controle são armazenados a cada estágio conforme a necessidade, assim como outras informações, como o registrador destino de escrita do banco de registradores, que deve ser enviado até o último estágio, aonde se realiza o armazenamento dos resultados. Na prática o pipeline não finaliza constantemente uma instrução por ciclo de relógio, frequentemente surgem situações em que uma determinada instrução em execução no caminho de dados depende de uma ainda não finalizada, o que nos leva a obter resultados indesejados. Essas ocorrências são denominadas de Dependências, ou Hazards, e podem ser divididas em Estruturais,

Figura 2.10 – Exemplo de execução do pipeline.



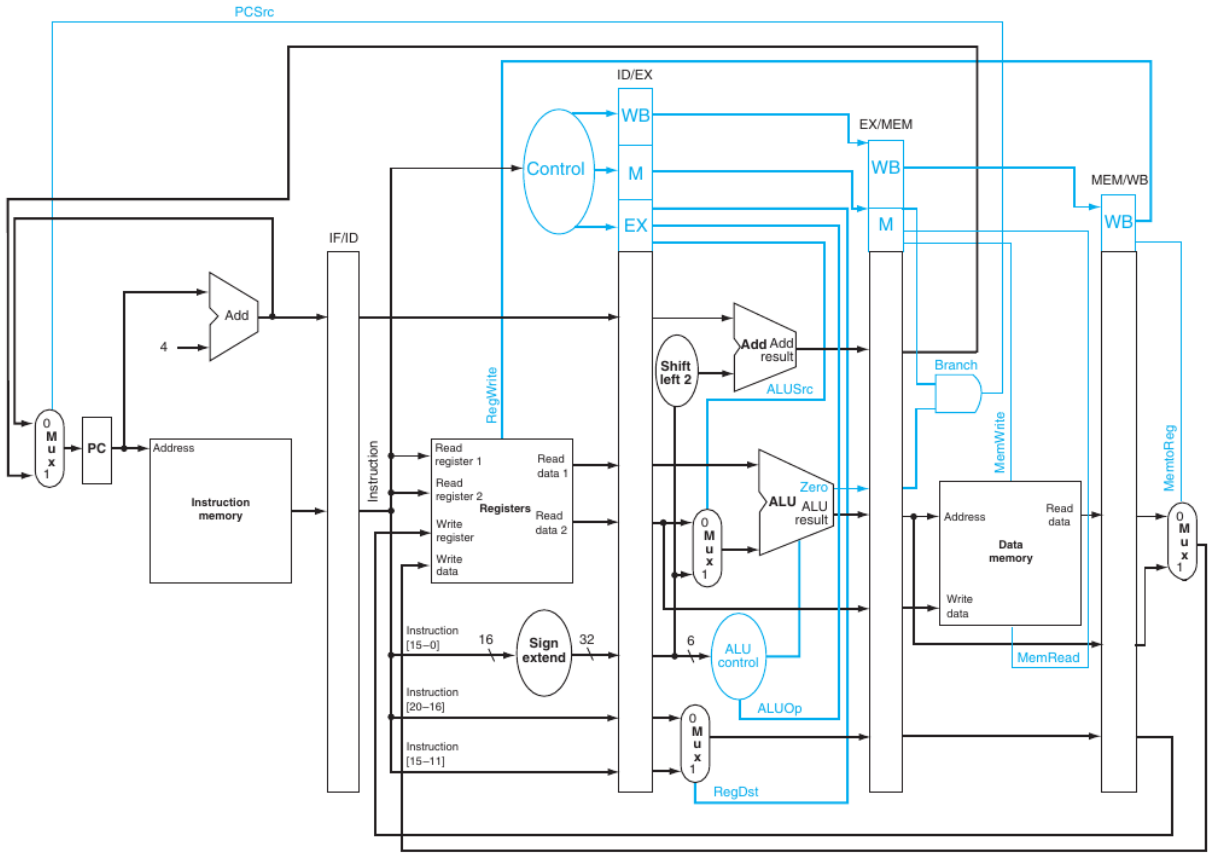
Fonte: Patterson e Hennessy (2004, p. 397).

Dados e Controle.

2.3.3 Dependências Estruturais

Ocorre quando um determinado componente do processador é acessado mais de uma instrução simultaneamente, por exemplo, no primeiro estágio quando se realiza a busca da instrução na memória de instruções, e o quarto acessa a memória de dados, temos que ambas utilizam do mesmo barramento para a troca de dados com a memória externa. Logo é necessário que o pipeline trave no estágio de busca, até que o quarto se complete, esse conflito pode ser resolvido com o uso de barramentos separados para a memória de instruções e a de dados, sendo essa divisão possível dentro da memória cache, por exemplo. O uso de barramentos separados para dados e instruções é denominado de Arquitetura Harvard, a Figura 2.10 indica essa ocorrência do acesso simultâneo da memória por dois estágios diferentes do pipeline, em um único ciclo de relógio, quando a primeira instrução (lw) está no estágio data acess, outra está sendo buscada no primeiro, Instruction fetch. Sem o uso de uma arquitetura harvard, necessitaríamos que o controle detecta-se essa dependência estrutural, travando o incremento de PC, e também inserido uma instrução que não altera-se nada no pipeline para preencher a que seria buscada da memória, como uma NOP. A característica de travar um estágio do pipeline, impedindo sua escrita, é denominado de "stall". Nessa situação, o tempo de execução da instrução atual passaria de um ciclo de relógio para dois, evidenciando a redução da performance do processador.

Figura 2.11 – Caminho de dados com pipeline de cinco estágios.



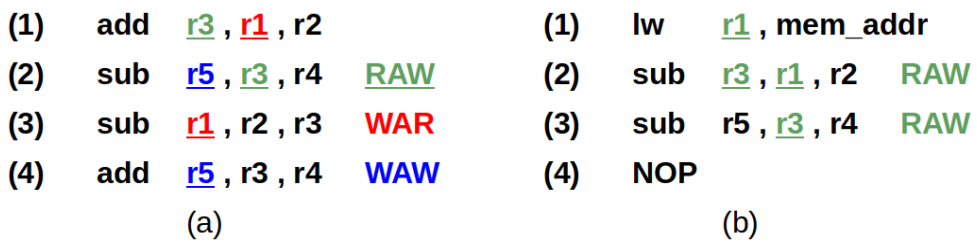
Fonte: Patterson e Hennessy (2004, p. 404).

2.3.4 Dependências de Dados

Indicam que uma instrução precisa do resultado de outra para poder ser executada, como exemplificam as instruções da Figura 2.12, em que a instrução dois utiliza o resultado da anterior como um operando, porém dentro do pipeline de cinco estágios, o resultado só estará disponível após o quinto ciclo de relógio, aonde se escreve no banco de registradores. Assim a instrução dois acaba por ler um valor diferente do desejado no segundo ciclo (etapa de leitura e decodificação). A essa dependência de dados se atribui o nome de RAW - Read After Write, conhecida também como dependência verdadeira, pode-se ainda citar a WAR - Write After Read, e a WAW - Write After Write, conhecidas como dependências falsas, porém essas ocorrem apenas quando se possuí uma arquitetura com a emissão e/ou conclusão de instruções fora da ordem (in-order ou out-of-order).

Uma análise sobre dependências é realizada por Pandey (2016), conceituando as dependências de dados em um pipeline de cinco estágios genérico, assim como demonstrando oito possíveis combinações de instruções que levam a sua ocorrência, sendo ressaltado o uso da técnica de Forward para solução do problema. Analisando a Figura 2.13 com a execução do código (a) da Figura 2.12, se percebe que apesar do resultado ser es-

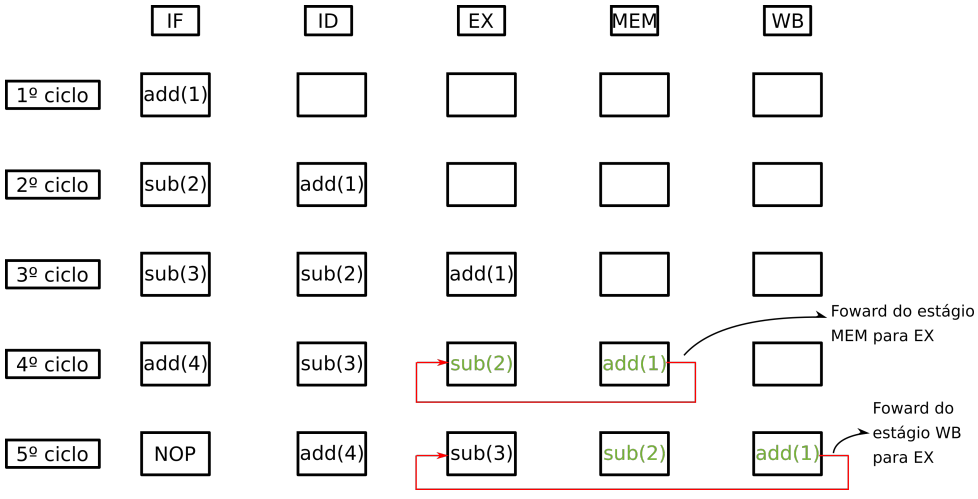
Figura 2.12 – Código em assembly com as ocorrências de RAW, WAR e WAW.



Fonte: Autor.

crito apenas após o quinto ciclo de relógio, o mesmo já está disponível a partir do quarto, quando a primeira instrução está no estágio MEM. Sendo é possível fazer uma realimentação desse valor para o estágio de execução como um operando da próxima instrução, caracterizando o Forwarding, que consiste em verificar se o endereço de escrita do estágio MEM é o mesmo do acessado por algum operando da ALU no estágio de Execução. Quando for verdadeiro o teste, a unidade de forward seleciona através de um multiplexador o sinal de realimentação como o correspondente operando da ALU, o mesmo se aplica caso o resultado esteja no estágio Write Back, logo temos a realimentação dos dois últimos estágios para o de Execute. Para evitar que a unidade de forward selecione desnecessariamente a realimentação como operando, é realizada também a verificação do sinal de escrita no banco de registradores dos estágios MEM e WB.

Figura 2.13 – Execução das instruções do código (a) da Figura 2.12 ao longo do pipeline. Todas as dependências foram resolvidas com a técnica de forwarding, não ocorrendo perdas de ciclo de relógio.

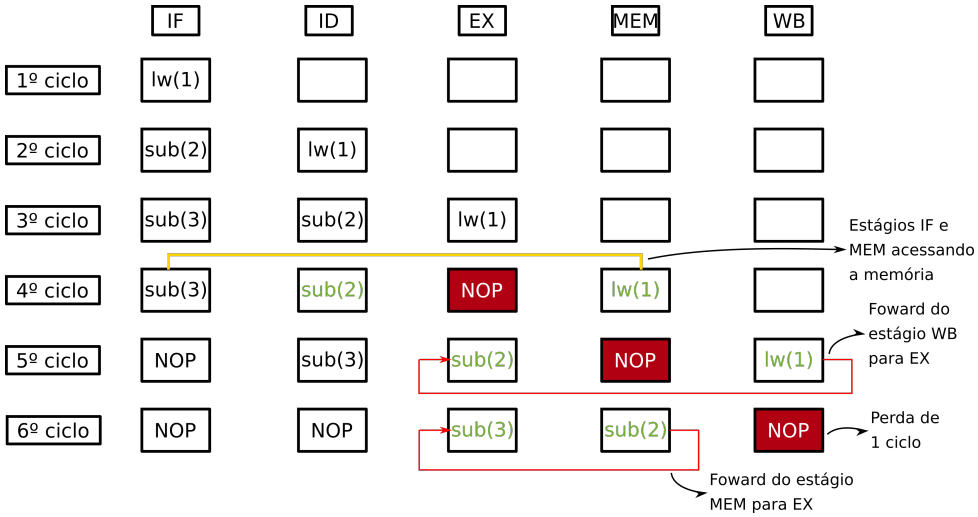


Fonte: Autor.

O caso citado para RAW consistiu apenas em instruções do tipo lógica/aritmética, porém com uma instrução LW seguida por uma que utilize o seu resultado, como no código (b) da Figura 2.12, se tem a perda de um ciclo de relógio, pois o seu resultado só estará

disponível no quinto estágio após a busca da memória, e a instrução a seguir precisa do seu valor no terceiro (Execução). Para isso é necessário que o controle detecte essa dependência de dados e insira um NOP (não executa nada) entre ambas, para no próximo ciclo realizar o forward do estágio WB para EX. Com instruções de escrita na memória, store word, não se tem perdas de ciclos, pois o valor a ser escrito deve estar presente no estágio de acesso a memória, o quarto, logo para ambas sequências lógica/aritmética com SW ou então LW com SW, temos que a técnica de forwarding é o suficiente para evitar perdas de ciclo de relógio. É importante notar que no caso de uma instrução no terceiro estágio possuir o mesmo operando de ambos os registradores destino do quarto e quinto estágio, o de maior prioridade é o mais recente, no caso o quarto estágio, a Figura 2.14 demonstra a execução do código (b) da Figura 2.12, com a implementação de forwarding, aonde se percebe a perda de performance devido a instrução LW seguida por uma lógica/aritmética. Foi considerado também que o processador não sofre de dependências estruturais de acesso a memória, apesar de ser destacado o momento em que ocorreria para referência.

Figura 2.14 – Execução das instruções do código (b) da Figura 2.12 ao longo do pipeline, com a técnica de Forwarding. No quarto ciclo surge a instrução NOP que o controle inseri no pipeline devido a dependência de dados entre lw(1) e sub(2).



Fonte: Autor.

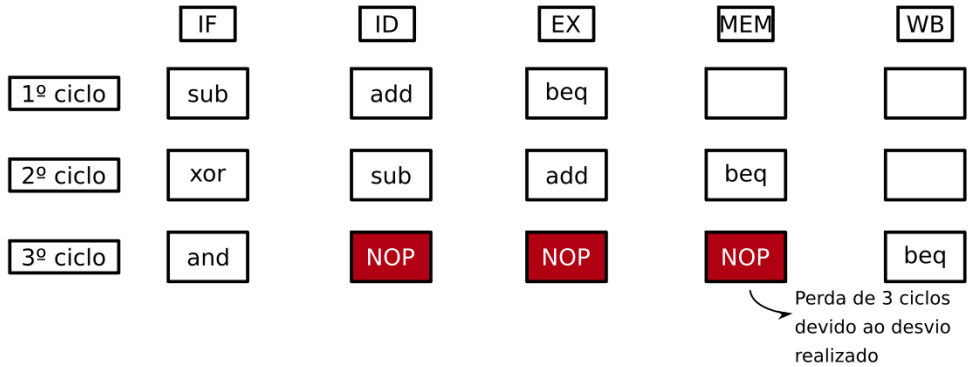
No trabalho por Kiat et al. (2017) se realiza uma analise sobre as dependências de dados de um processador RISC com um pipeline de cinco estágios. O processador em questão possuí os estágios Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Data Memory Access (MEM) e Write Back (WB). A sua proposta envolve na alteração do caminho que a técnica de forwarding utiliza, a fim de resolver as dependências de dados, essa mudança consiste em trazer um estágio para traz a técnica, ou seja, quando antes se conectava o sinal de forward dos estágios MEM e WB, para o estágio EX, agora se conecta o sinal dos estágios EX e MEM, para o estágio ID. Essa modificação beneficiou-os com dois

pontos, um que o circuito combinacional da ALU possuia uma latência maior no estágio EX, do que o banco de registradores no estágio ID, dessa forma ajuda a equilibrar a divisão de tempo do pipeline. A outra vantagem envolve a redução de sinais de controle que percorrem o caminho de dados. O trabalho também exemplifica a ocorrência de diversos casos da ocorrência de dependências de dados, tanto para a abordagem proposta, como a utilizada como base para modificação, de Patterson e Hennessy (2004).

2.3.5 Dependências de Controle

Considerando o pipeline de cinco estágios, quando se tem a execução de uma instrução branch, desvio condicional, o resultado do teste para desvio de endereço é conhecido apenas no quarto estágio, porém o pipeline está a cada ciclo buscando novas instruções da memória, mesmo sem saber se as mesmas devem ser executadas ou não, de acordo com o resultado o branch. Caso se tenha que o desvio deve ser realizado, todas as instruções que já estavam em andamento no pipeline, no caso três, do primeiro ao terceiro estágio, devem ser descartadas, causando uma perda de três ciclos, a esse problema nos referimos como dependências de Controle (PATTERSON; HENNESSY, 2004). No caso da execução de desvios incondicionais (jump), sempre ocorre a perda de ciclos, a unidade de controle deve detecta-lo e inserir instruções NOP no pipeline. A ação de descartar o pipeline é muitas vezes dita como "flush". Na Figura 2.15 se demonstra as perdas por dependências de controle.

Figura 2.15 – Execução de uma instrução branch com desvio tomado em um pipeline de cinco estágios.

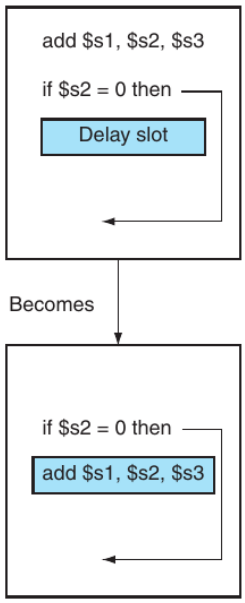


Fonte: Autor.

Diferentes técnicas foram criadas a fim de reduzir a perda drástica de desempenho devido a desvios, pois o aumento do número de estágios de pipelines para uma maior frequência de sinal de relógio, também implica em maiores perdas de desempenho por dependências de controle. Um meio consiste em inserir instruções após o branch que

sempre serão executadas, dessa forma ignora-se completamente se o mesmo irá realizar ou não o desvio, nota-se que a carga fica dessa forma sobre o compilador, e não o hardware, pois o mesmo deve encontrar instruções que não possuem relação com o branch, e que possam ser executadas antes do mesmo, a fim de as mover de lugar. A essa técnica atribui-se o nome de Branch Delay Slot, apesar de eliminar as dependências de controle, não é eficiente, pois em códigos grandes temos uma grande cadeia de dependências não só de controle, mas também de dados, o que torna inviável para um compilador executar tal tarefa. Outra abordagem envolve minimizar o impacto dos desvios, para isso se adianta o cálculo do endereço de desvio, e se ele é realizado ou não, para os primeiros estágios do pipeline, porém como dito, isso apenas minimiza os efeitos, e implica em aumento do hardware, pois necessitaria de componentes dedicados para realizar essa tarefa com antecedência. É demonstrado na Figura 2.16 a alteração na ordem das instruções, definindo o uso do Branch Delay Slot.

Figura 2.16 – Branch delay slot, reposiciona a instrução add após o desvio condicional, pois a mesma não afeta na sua decisão, logo pode ser executada sempre.



Fonte: Adaptador de Patterson e Hennessy (2004, p. 424).

Outra forma de atenuar as perdas por instruções de branch se da pela tentativa de prever o resultado do teste da instrução antes mesmo de ela chegar ao estágio decisivo, se acertar, as perdas são nulas, do contrário realiza o procedimento de limpar o pipeline das instruções em execução nos estágios anteriores, e atualizar o PC para o seu novo valor (endereço de desvio). As previsões são divididas em duas categorias, previsão estática e dinâmica, sendo comum se referir ao estado da previsão como taken (desvio tomado) e not taken (desvio não tomado). Na previsão estática, o processador vai sempre assumir que o desvio é taken ou not taken, e caso o quarto estágio prove que a previsão foi errada, se tem

a perda de três ciclos de relógio, do contrário se mantém a performance de uma instrução por ciclo. Geralmente é utilizada a política not taken, pois a fim de implementar a taken como padrão estático, é necessário saber o endereço da instrução a ser buscada logo no próximo ciclo após a busca da instrução branch da memória, sendo que seu endereço alvo ainda não foi calculado. Sendo taken ou not taken, uma vez implementada a técnica, não pode ser mudada, a decisão é realizada no momento de projeto e implementação do hardware e assim permanecerá. Sua utilização possui como ponto positivo o fato de, para a política not taken, não acrescentar tamanho ao hardware, mantendo a simplicidade da execução de instruções branch, porém é ineficiente em muitos casos, como em códigos que contém loops por exemplo, ou em que os desvios condicionais oscilam muito entre taken e not taken.

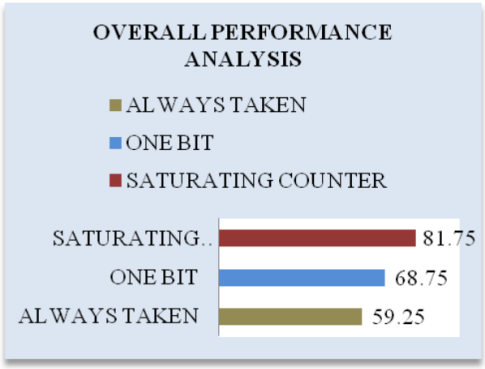
Para a previsão dinâmica temos que o processador altera constantemente a tentativa de prever o resultado entre taken e not taken, tal decisão é baseada em diferentes aspectos, como por exemplo, em um histórico que indica a probabilidade da instrução de branch possuir o teste verdadeiro ou falso de desvio. Uma abordagem simples dessa técnica consiste em verificar o endereço da instrução branch buscada da memória, que corresponde ao valor de PC, seu comportamento será então previsto de acordo com a sua última execução. Para isso se necessita de uma forma de armazenar as informações das instruções de desvio quando executadas, utilizando-se de uma Branch History Table - BHT, que consiste em uma memória capaz de armazenar um endereço de instrução branch, o endereço alvo da instrução, e um bit de previsão de desvio. Quando uma nova instrução branch é buscada no estágio IF a mesma possui o seu endereço comparado com a BHT, se houver registros dela, verifica-se então qual o valor lógico do bit de previsão, para 0 se prevê como not taken, e segue o fluxo normal de incremento de PC, quando 1, supõe-se como taken e atualiza o valor do PC imediatamente no próximo ciclo para o endereço alvo contido na mesma linha acessada da BHT. Como se percebe, é necessário um grande número de comparadores nessa abordagem, tornando o hardware preditor excessivamente grande, para isso é comum utilizar apenas os bits menos significativos do endereço para indexar a BHT, e o restante é utilizado como uma Tag, para validar se a linha acessada da BHT de fato é uma instrução branch equivalente.

Da mesma forma que na previsão estática, se o preditor errar as instruções são descartadas, e então se complementa o bit de previsão da BHT, mudando o chute que o preditor irá realizar na próxima vez que receber esse mesmo branch. Ao iniciar o sistema, todas os branch são considerados como not taken devido a falta de informação, pois nenhuma instrução foi executada ainda para se coletar informações comportamentais de previsão. É também utilizado um bit de validade, para indicar quando uma determinada linha da BHT possui informações validas ou não, a fim de evitar que o preditor realize desvios sobre endereços que não deveriam. Como um hardware com custo alto dentro do processador, o seu tamanho é limitado, logo ao ser totalmente preenchido se tem a ne-

cessidade de determinar como será realizada as escritas sobre o mesmo quando um novo branch for executado pelo pipeline, dentre essas técnicas temos disponíveis o uso de uma FIFO (First-in First-out), LUR (Last Used Recently), LFU (Last Frequently Used), ou então dependendo da forma de mapeamento da BHT, não se tem algoritmos de substituição, como no mapeamento direto (ou associativo por conjunto one-way).

É perceptível a diferença de flexibilidade entre a previsão estática e a dinâmica, na primeira se tem a simplicidade de se implementar, necessitando esforços do hardware apenas em caso de implementação da política taken, em que surgem de hardwares dedicado para cálculo prévio do endereço de desvio, em comparação na dinâmica a complexidade do hardware é elevada a uma memória de armazenamento interna no processador, exclusiva para o histórico ou comportamento das instruções de branch, incluindo até mesmo máquinas de estados para controlar o seu funcionamento. Porém a dinâmica possuí uma taxa de acertos muito superior a estática, Arora, Kotecha e Samyal (2013) analisaram o desempenho da implementação estática taken, dinâmica de 1 bit e dinâmica do contador por saturação, e a performance geral foi respectivamente de 59,25%, 68,75% e 81,75%, conforme a Figura 2.17, aonde se evidencia a diferença de flexibilidade entre os diferentes preditores.

Figura 2.17 – Desempenho geral dos preditores.

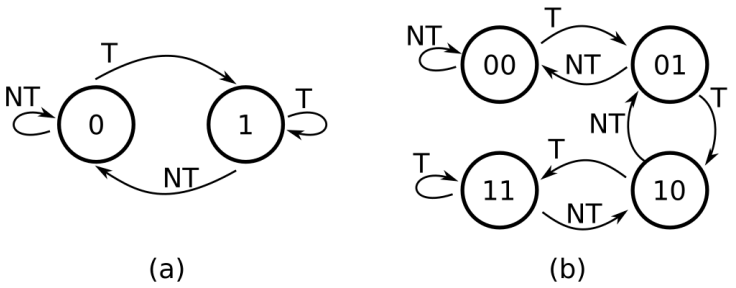


Fonte: Arora, Kotecha e Samyal (2013).

Nos testes com previsão estática taken, assim que a instrução é decodificada e o endereço alvo é decodificado, se realiza o desvio para o endereço alvo, e então caso venha a errar a previsão, se descarta o pipeline e corrige o valor do PC. O preditor de 1 bit utilizado consiste em uma memória indexada pelos MSb do endereço da instrução branch, quando é encontrada uma instrução na memória, se verifica o bit de desvio e realiza o desvio de acordo com, se o preditor errar, descarta o pipeline e corrige o valor do bit de previsão. Por fim, o terceiro caso analisado no trabalho consiste no Contador por Saturação, que funciona de forma semelhante ao de 1 bit, porém agora se utiliza 2 bits para a previsão, e seu comportamento é determinado por uma máquina de estados finitos. O MSb determina se é taken ou não, ou seja, para 11 ou 10, se prevê a instrução como taken, do contrário,

not taken, quando o preditor erra, a alteração nos bits de previsão ocorre de acordo com o seu estado atual, que pode ser Strongly Taken - 11, Weakly taken - 10, Weakly Not Taken - 01, e Strongly Not Taken - 00. A Figura 2.18 demonstra a lógica de atualização da previsão para 1 e 2 bits.

Figura 2.18 – Funcionamento da previsão para 1 e 2 bits, (a) e (b) respectivamente.



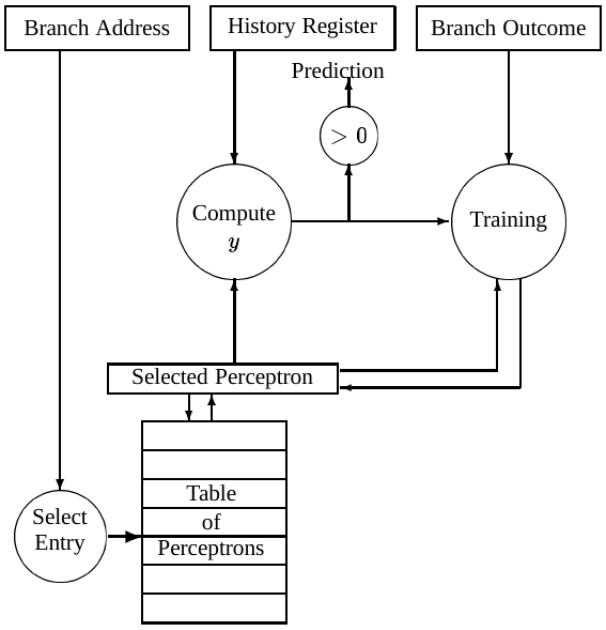
Fonte: Adaptado de Arora, Kotecha e Samyal (2013).

Jimenez e Lin (2001) demonstra um método de predição dinâmico alternativo ao do contador de 2 bits, por meio de uma rede neural perceptron simplificada. Sua escolha pela perceptron se deu pela sua eficiência de implementação em hardware, provendo uma troca relativamente boa de custo do hardware, por precisão. Outro fator que levou ao uso de perceptrons é sua simplicidade na tomada de decisões, seguindo equações matemáticas, diferente de outros tipos de redes neurais que nem sempre possuem uma transparência sobre suas ações, tornando difícil de compreende-las. A Figura 2.19 o esquemático do preditor perceptron, o sinal "Branch address"é utilizado para para selecionar uma perceptron que é lida da tabela. A perceptron selecionada, em conjunto com o "History Register", processa a decisão de desvio condicional, sendo então realizada a atualização da perceptron com o algoritmo de treinamento, e sua escrita de volta a tabela. Sua vantagem em relação ao contador de 2 bits se dá por conta da possibilidade de utilizar maiores históricos para o preditor, com um aumento significativamente menor de recursos, mesmo que sua complexidade seja relativamente elevada em comparação ao contador bimodal.

2.4 FPGA

Introduzidos por volta de 1970s, os Programmable Logic Devices (PLDs) consistem em chips de propósitos gerais em implementação de circuitos lógicos. Podem ser vistos como uma "caixa preta"que contém portas lógicas e conexões programáveis, montando assim a lógica desejada (BROWN; VRANESIC, 2008, p. 98). Um tipo de PLD com grande capacidade lógica são os Field-Programmable Gate Array (FPGA), que em sua estrutura geral possuem "blocos lógicos"que implementam as funções inseridas, "blocos IO"que co-

Figura 2.19 – Diagrama em blocos do preditor de desvios dinâmicos perceptron.



Fonte: Jimenez e Lin (2001).

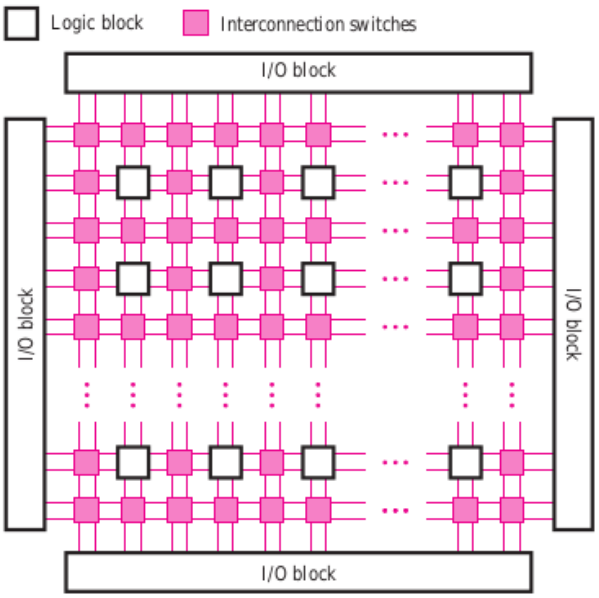
nectam os pinos do encapsulamento e conexões internas, conforme o esquemático da Figura 2.20.

2.4.1 Placa de Desenvolvimento Nexys3

A plataforma desenvolvimento da Nexys3 conta com a FPGA Xilinx Spartan-6, dispositivo XC6SLX16, proporcionando uma capacidade lógica elevada para projetos digitais, e também outros componentes que ajudam na implementação de sistemas mais complexos, como se deixa explicito na Figura 2.21. Na placa se tem disponível um oscilador CMOS de 100 MHz, 16 MBytes de Cellular RAM, memória paralela PCM não volátil de 16 MBytes, memória serial PCM não volátil de 16 Mbytes, porta VGA de 8 bits, 4 displays de 7 segmentos, entre outros componentes (DIGILENT, 2013). É destacado também duas portas de comunicação USB, uma com a função de carregar a configuração da FPGA, Adept USB Port, e a outra para uso da UART, USB UART. Sua alimentação pode ser proveniente da própria porta Adept USB Port, ou então de uma fonte externa, sendo no caso a alimentação USB o suficiente para inúmeros projetos com baixo consumo de potência.

As memórias cellular RAM e PCM paralela da placa compartilham os barramentos de dado e endereço, com tamanho de 16 e 24 bits respectivamente, o mesmo se aplica aos seus sinais de habilitação de saída e escrita de dados, OE (output enable) e WE (write enable), porém seus sinais CE (chip enable) são separados. Na Figura 2.22 se demonstram os sinais das memórias celular RAM e PCM paralela.

Figura 2.20 – Estrutura geral de uma FPGA.



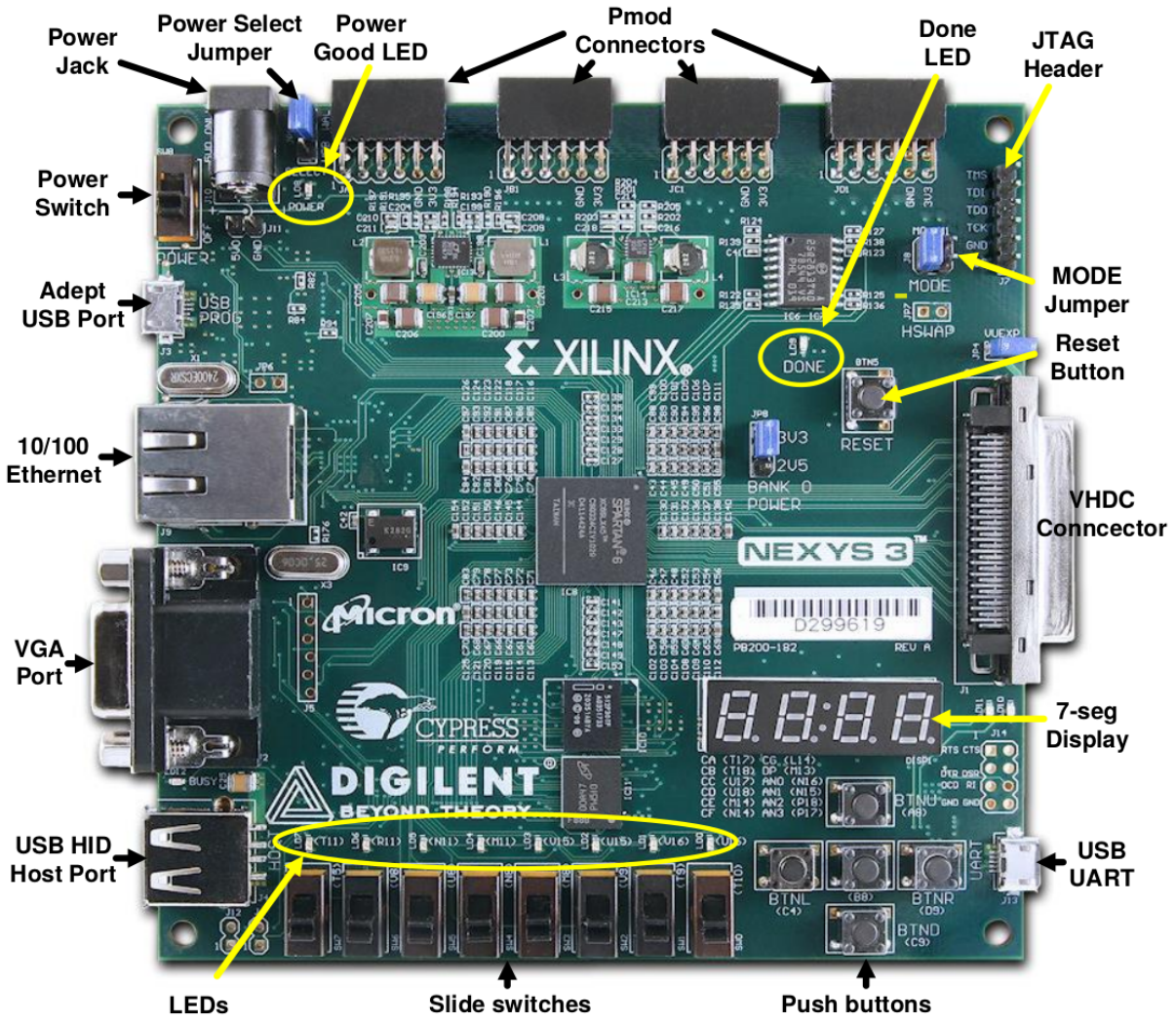
Fonte: Brown e Vranesic (2008, p. 110).

Na memória cellular RAM ainda temos o sinal MT-CLK (sinal de clock), MT-WAIT (wait), MT-ADV (address valid) e MT-CRE (control register enable), para serem utilizados com a memória no modo síncrono, com ela operando nesse modo se obtém um barramento com uma frequência de até 80 MHz. Quando sua operação está no modo assíncrono, o período mínimo é de 70 ns para um ciclo de leitura/escrita, equivalente a uma frequência máxima de 14,2 MHz, também nota-se que nesse modo a memória realiza automaticamente o refresh das DRAMs simplificando o seu controle. O endereçamento é a byte, logo é possível acessar apenas 8 bits dos 16 acessados em uma operação de leitura ou escrita, para isso se tem os sinais de controle MT-UB (upper byte) e MT-LB (lower byte).

Já na PCM paralela temos um endereçamento a palavra de 16 bits, ou seja, o acesso a cada endereço realiza necessariamente a leitura ou escrita de 16 bits. Com um total de 16 Mbytes de memória, se tem então 8Mwords de 16 bits, também se tem na PCM um sinal de Reset (RP#). Contém 128 blocos individuais apagáveis de 64K, que podem ser subdivididos em 4 blocos de 16K. A memória oferece um tempo de acesso para leitura de 115 ns de período, para leitura, e 25 ns no page-mode. O dispositivo PCM serial possui um barramento de 50 MHz. Ambos os dispositivos PCM podem ser utilizados para escrita de arquivos de configuração da FPGA, que configurados, causam com que a FPGA faça a leitura da configuração ao ser energizada. Um arquivo de configuração da FPGA Spartan-6 XL16 ocupa 512 Kbytes, deixando livre para uso aproximadamente 97% de sua capacidade de armazenamento (DIGILENT, 2013, p. 9).

A comunicação serial da FPGA se da por meio do circuito integrado FTDI FT2 USB-UART bridge, que é conectada conforme o esquemático da Figura 2.23. Dados são tro-

Figura 2.21 – Plataforma de desenvolvimento Nexys3.



Fonte: Digilent (2013, p. 3).

cados de forma serial através das conexões TXD e RXD, com uso de um software de computador adequado para a comunicação.

2.4.2 Spartan-6 XL16

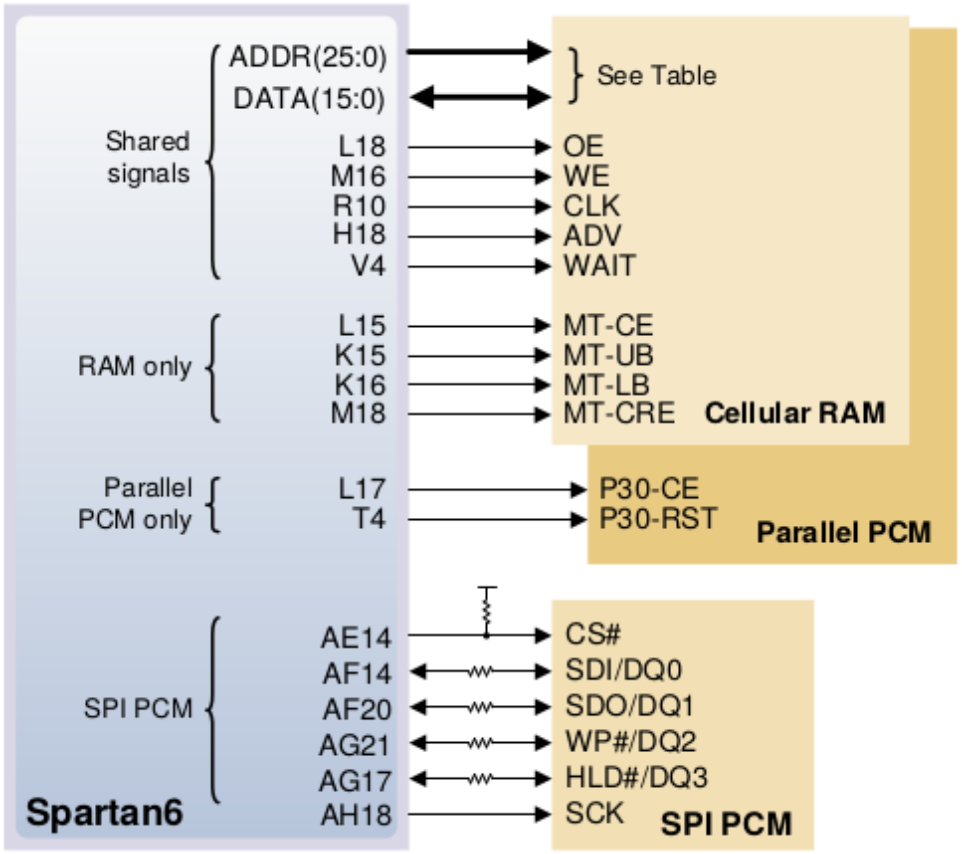
A FPGA da Xilinx Spartan-6 XL16 conta com as seguintes características:

• 2.278 slices, cada um contendo 6-input LUT (look-up-table) com 8 flip-flops, totalizando 18.224 flip-flops;

• 576 Kbits de blocos de memória RAM rápidas, BRAM;

• 32 BRAM de 18 Kb;

Figura 2.22 – Conexões entre a FPGA e as memórias cellular RAM, PCM paralela e PCM serial.



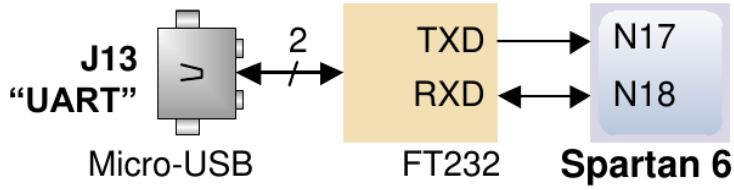
Fonte: Digilent (2013, p. 10).

- 32 DSP slices;
- Máximo de 232 I/O para usuário.

O uso da DSP se encontra em multiplicações binárias e acumuladores, que possuem uma alta performance de velocidade e baixo consumo de potência. Cada DSP slice contém um multiplicador a complemento de 2 dedicado de 18 x 18 bits e um acumulador de 48 bits. Operando de forma síncrona, pode ser utilizado com múltiplos estágios de registradores, a fim de aumentar sua performance geral com pipeline. O multiplicador também oferece a possibilidade de executar uma operação de barrel shifting (XILINX, 2011a).

Os blocos de RAM do dispositivo operam de forma síncrona tanto na leitura quando na escrita, e podem ser utilizadas com diferentes configurações, como somente leitura, escrita e leitura e ROM, por exemplo. Cada um dos blocos podem ser utilizados como somente um único de 18K bits de dados, ou então como dois blocos de 9K bits independentes. Assim como no uso da DSP, é possível adicionar níveis de registradores para aumentar a performance do mesmo por meio da técnica de pipeline (XILINX, 2011b). Os blocos podem ser configurados como single-port ou dual-port, que se referem ao uso de

Figura 2.23 – Conexões entre a FPGA e o dispositivo para comunicação serial UART.



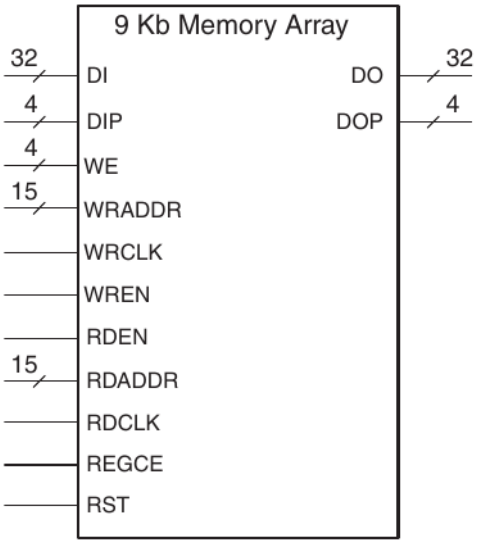
Fonte: Digilent (2013, p. 12).

uma porta ou duas da RAM. No modo de operação dual-port, as portas da RAM são simétricas, compartilhando dos mesmos dados armazenados, porém fazendo acessos independentes, o que pode ocasionar em colisões. Quando utilizados os blocos de RAM, o tamanho do endereço de acesso em bits pode variar, de acordo com o tamanho da memória desejada, porém no caso de não se ocupar toda a capacidade do bloco (9 ou 18 Kbits), o restante dos bits de endereçamento são concatenados com 0. Sua inicialização pode ser feita por meio do bitstream de configuração, não dispondo da possibilidade do uso de sinais de reset para a memória, mas é possível utilizar o sinal de reset nos registradores de saída da memória (utilizados no pipeline).

Quando se usa uma BRAM como single-port, a mesma pode realizar tanto leitura quando escrita de forma síncrona, já em uma abordagem dual-port surgem duas possibilidades, a simple-dual-port e a true-dual-port. Para a simple-dual-port BRAM temos que uma das portas é responsável apenas pela leitura, e a outra é utilizada para a escrita de dados, assim não há colisão de dados. Ambas as portas possuem entradas próprias de endereço de acesso, e também de sinal de relógio para funcionamento síncrono. A Figura 2.24 contém um diagrama de uma simple-dual-port BRAM. Já para a true-dual-port se tem que ambas as portas são capazes de realizar tanto escrita quanto leitura, o que pode gerar conflitos de dados quando ambas tentam escrever no mesmo endereço, por exemplo.

Os diversos recursos disponíveis na FPGA podem ser utilizadas basicamente de quatro formas, instanciados, inferidos, CORE Generator & Wizards e por Macro. O uso da instanciação pode ser feito diretamente na descrição, sendo útil quando se deseja controlar o local exato do bloco individual, já para inferir se tem a necessidade do suporte da ferramenta de síntese, que é bem comum. A vantagem de inferir um componente está no fato da portabilidade e flexibilidade do código para diferentes arquiteturas. Outro ponto positivo também está no fato das ferramentas poderem otimizar o seu uso para performance, área, ou energia para o usuário, pela ferramenta de síntese. O CORE Generator & Wizards são formas gráficas de gerar os componentes, porém não possuem portabilidade, e surge a necessidade de refaze-lo ao transferir para diferentes arquiteturas, uma vantagem está na facilidade para implementação de blocos mais largos. O uso de Macro se da por meio da UniMacro Library das ferramentas da Xilinx, sendo utilizadas para instanciar componentes primitivos que são muito complexos para instanciar de forma convencional (XILINX, 2013,

Figura 2.24 – Diagrama de blocos de simple-dual-port BRAM. DO - Data Output Bus, DOP - Data Output Parity BUS, DI - Data Input Bus, DIP - Data Input Parity Bus, RDADDR - Read Data Address Bus, RDCLK - Read Data Clock, RDEN - Read Port Enable, REGCE - Output Register Clock Enable, RST - Synchronous Set/Reset the output registers/latchers, WE - Byte-wide Write Enable, WRADDR - Write Data Address Bus, WRCLK - Write Data Clock, WREN - Write Port Enable.



Fonte: Xilinx (2011b, p. 17).

p. 4).

É possível definir restrições de tempo, timing constraints, a fim de se alcançar objetivos estabelecidos em projetos de alta performance. O guia por Xilinx (2009a) estabelece os fundamentos das restrições, PERIOD CONSTRAINTS, OFFSET CONSTRAINTS e FROM:TO CONSTRAINTS. As restrições de tempo ainda podem ser divididas de acordo com o caminho a ser coberto pela ferramenta de análise, sendo os caminhos mais comuns:

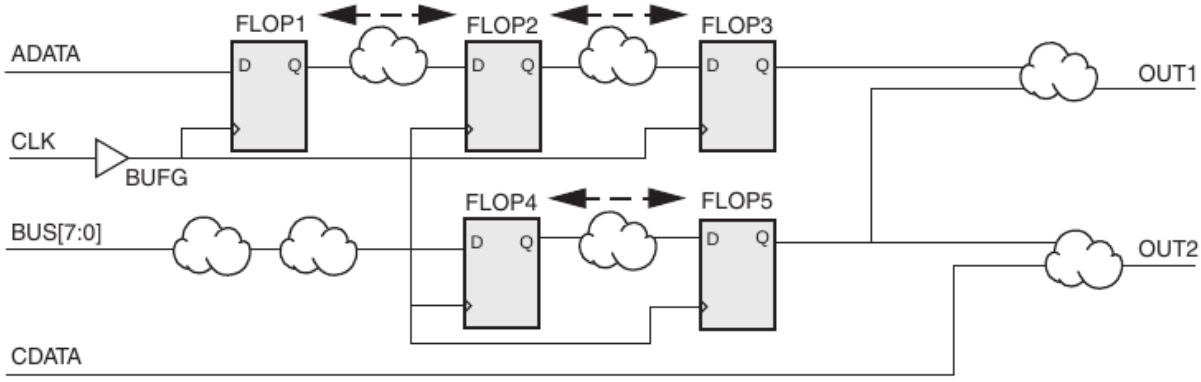
- Caminhos de entrada;
- Elemento síncrono para elemento síncrono;
- Exceções de caminhos específicos;
- Caminhos de saída.

As restrições dos caminhos de entrada, input timing constraints, cobrem o tempo de propagação de um sinal desde um pino de entrada externo da FPGA, até um elemento de memória (FF, Registrador) interno do projeto, que adquire o dado. A constraint usada para especificar tal parâmetro é a OFFSET IN, que analisa a relação entre o dado e a borda de sensibilidade do sinal de relógio, assim, quando a ferramenta de tempo realiza a analise, é considerado outros fatores como frequência, fase e incertezas do sinal de relógio, e também ajustes de latência do dado. O OFFSET IN está associado a apenas

uma entrada de sinal de relógio, cobrindo todos os caminhos das entradas de dados até elementos síncronos pelo mesmo sinal estabelecido, esse método é chamado de global, sendo a forma mais eficiente de se especificar temporizações da entrada (XILINX, 2009a, p. 13).

Nas restrições entre elementos síncronos, register-to-register timing constraints, se tem a latência estabelecida do caminho lógico combinacional entre dois elementos de memória, a global constraint PERIOD considera aspectos como os requerimentos de tempo do domínio de sinal de relógio, o caminho em um único ciclo, todos os caminhos relacionados ao sinal de relógio, e também diversos fatores do sinal de relógio como fase e incertezas. A constraint PERIOD cobre apenas caminhos entre dois elementos de memória, descartando os pinos de entrada e saída da FPGA, conforme demonstra a Figura 2.25, também considera o tempo entre elementos síncronos utilizando recursos da FPGA, como DSPs ou RAMs, por exemplo.

Figura 2.25 – Exemplo do caminho analisado pela constraint PERIOD.



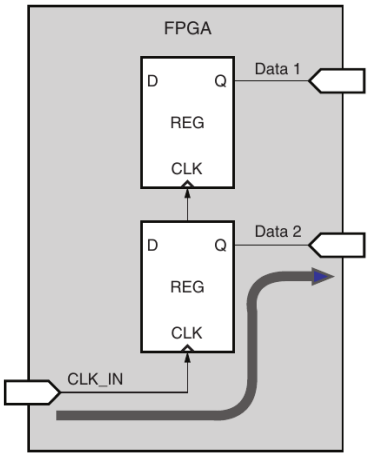
Fonte: Xilinx (2009a, p. 47).

Para os caminhos de saída, output timing constraints, temos a cobertura da transmissão de um sinal de um elemento de memória, para um pino de saída externo da FPGA. A constraint OFFSET OUT é utilizada para especificar a temporização de saída, estabelecendo assim o tempo máximo de transmissão da FPGA para um pino externo. A latência começa na propagação do sinal de relógio no pino de entrada até o elemento de memória, que transmite então na borda de sensibilidade o sinal de sua saída, até o correspondente pino externo da FPGA. A Figura 2.26 demonstra a cobertura do caminho pela ferramenta de analise.

Na Figura 2.27 se obtém o uso das três global constraints, a fim de se estabelecer uma frequência de trabalho mínimo, em um circuito elementar.

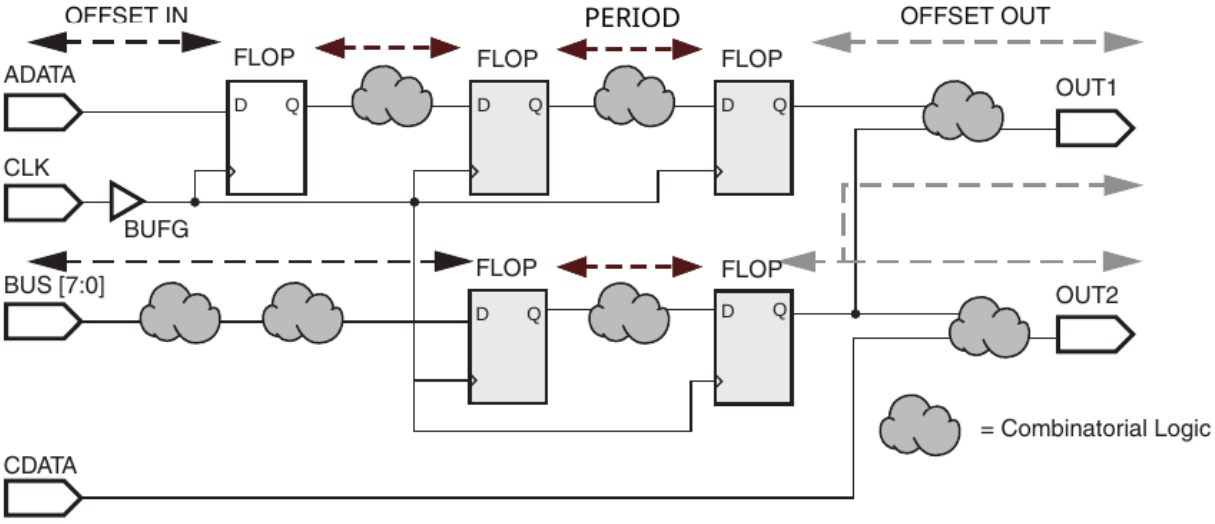
O uso das constraints globais geralmente cobrem quase todos os caminhos a serem analisados, porém uma pequena parcela pode necessitar de uma regra em específica, sendo essas exceções os caminhos falsos e/ou multicíclicos. Os caminhos falsos

Figura 2.26 – Restrição de tempo da saída, da entrada de sinal de relógio, para o elemento síncrono, e então para a saída de dados do dispositivo.



Fonte: Xilinx (2009a, p. 21).

Figura 2.27 – Visão dos caminhos analisados pelas global constraints: OFFSET IN, PERIOD e OFFSET OUT.



Fonte: Adaptado de Xilinx (2009a, p. 53).

consistem na remoção de caminhos específicos da analise de tempo, para isso se utiliza a constraint FROM-TO com a palavra chave timing ignore (TIG). Em caminhos multicíclicos dados são transferidos entre elementos síncronos a uma frequência diferente da definida por PERIOD. As timing constraints podem ser estabelecidas no projeto de duas formas, pela descrição HDL, ou pelo Constraint Editor (UCF) do software ISE Design, maiores detalhes sobre o uso de timing constraints podem ser encontrados no manual por Xilinx (2009a).

2.5 TRABALHOS RELACIONADOS AO RISC-V

Nesta seção se apresenta alguns trabalhos já realizados sobre a ISA do RISC-V. Se buscou arquiteturas próximas das projetadas neste trabalho, incluindo assim núcleos com a técnica de pipeline, uso de bases e extensões do RISC-V, e linguagens de descrições de hardware. A Tabela 2.2 demonstra uma comparação dos trabalhos aqui verificados, em relação ao abordado neste projeto.

Tabela 2.2 – Comparação geral entre o projeto e outros trabalhos já realizados sobre a ISA do RISC-V. O número de estágios de cada arquitetura pipeline está indicado, assim como os casos em que se implementou diferentes profundidades do mesmo.

Núcleo	ISA	HDL	Dispositivo
	Projeto		
Pipeline (5)	RV32EC	VHDL	Xilinx Spartan-6
Multicíclico	RVC	VHDL	Xilinx Spartan-6
	Trabalhos Relacionados		
Monocíclico	RV32I	Verilog	Xilinx Spartan-3
Pipeline (5)	RV32IF	SystemVerilog	Xilinx Virtex-6
Pipeline (2/3/4)	RV32IM	—	Xilinx Série 7
Taiga: Pipeline	RV32IMA	SystemVerilog	Intel e Xilinx
ORCA: Pipeline (4/5)	RV32I/RV32IM	VHDL	—
RISCY: Pipeline (4)	RV32IMFC	—	ZedBoard
Zero-RISCY: Pipeline (2)	RV32IMC/RV32EMC	—	ZedBoard

Fonte: Autor.

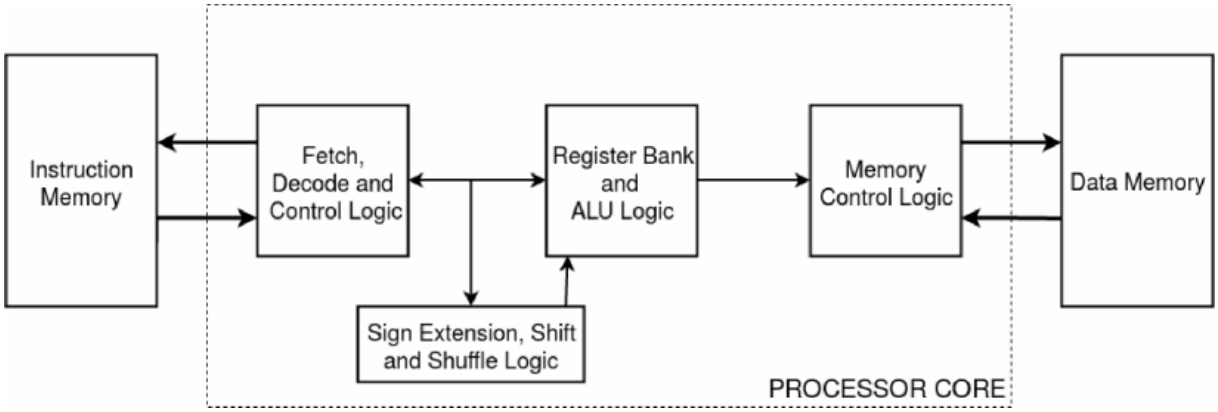
Dos trabalhos revisados, nenhum faz uma abordagem de um núcleo multicíclico, que neste trabalho se optou também por utilizar nele apenas as instruções da extensão C, sem nenhuma das bases do RISC-V. Para o núcleo pipeline seguiu-se o modelo clássico de 5 estágios, porém com a base E, e não a I, implementada pela grande maioria dos trabalhos. A plataforma PULPino é a única a utilizar da base E, porém com um pipeline reduzido de apenas 2 estágios, e com instruções de multiplicação (extensão M).

2.5.1 Análise de Processador Monocíclico

No trabalho por Dennis et al. (2017), se implementou um processador monocíclico em uma FPGA Xilinx Spartan-3 XC3S500E, capaz de executar as instruções da base I do RISC-V, com instruções de 32 bits, RV32I, a descrição foi realizada em Verilog HDL, utilizando o software Xilinx ISE 14.7 para síntese e implementação. Em uma visão de alto nível, se dividiu o núcleo em quatro etapas lógicas, conforme demonstra a Figura 2.28, em que o bloco lógico Fetch, Decode and Control Logic realiza a busca da instrução da memória, também realizando o cálculo de endereços de branch. Possui um somador dedicado

para incrementar o valor de PC, uma unidade decodificadora puramente combinacional. O bloco Register Bank and ALU Logic possui internamente os 32 registradores de uso geral da ISA do RISC-V, e a unidade de operações lógicas/aritméticas do núcleo. Para operações register-immediate, a ALU é provida com um dos operando pelo bloco Sign Extension, Shift and Shuffle Logic, que realiza a decodificação do sinal imediato que está embaralhado na instrução, realizando então shift nos lsb quando necessário, para evitar desalinhamento de acesso a memória. Os resultados de implementação do processador monocíclico demonstraram que a frequência máxima de operação obtida foi de 32 MHz, consumindo um total de 7,9m W de potência, e utilizando um total de 5578 LUT.

Figura 2.28 – Visão de topo do núcleo monocíclico implementado por Dennis et al. (2017).



Fonte: Dennis et al. (2017).

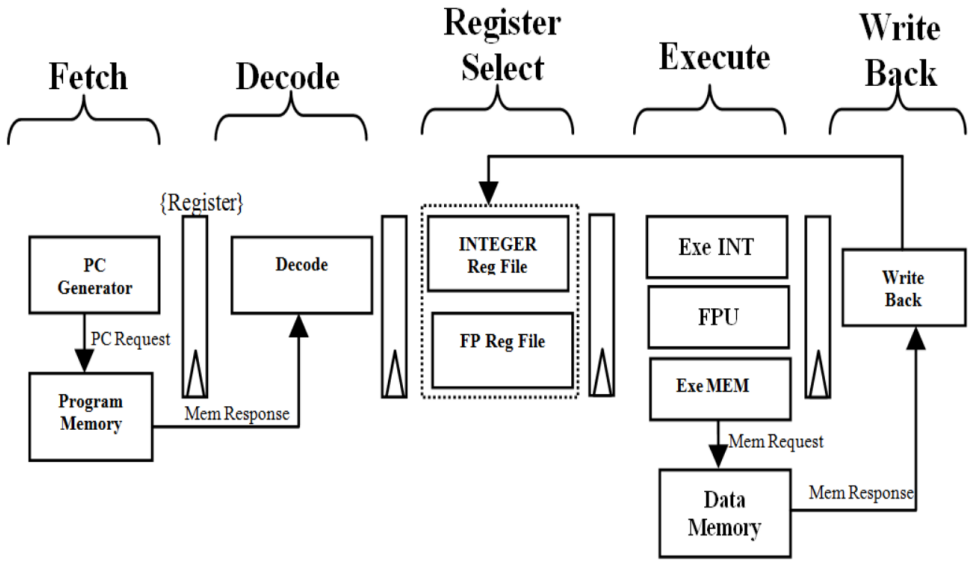
2.5.2 Análise de um Processador Pipeline de 5 Estágios

Na publicação por Raveendran et al. (2016) se tem uma análise sobre a influência do conjunto de instruções do RISC-V, no projeto de um processador com pipeline de 5 estágios, utilizou-se de SystemVerilog na simulação e sínteses do núcleo em uma FPGA Xilinx Virtex-6 xc6vlx550t-2ff1759. Também se sintetizou o projeto com uma tecnologia de 65nm e 130nm para ASIC. As características do núcleo consistem em emissão de uma única instrução, unidade de ponto flutuante, e implementação da base I do RISC-V com a extensão de instruções em ponto flutuante, F. Os estágios do pipeline se dividem, em ordem de execução, Fetch, Decode, Register Select, Execute, Write Back. No primeiro estágio se realiza o acesso a memória para busca da instrução, sendo no segundo realizado a sua decodificação, as informações obtidas no segundo estágio então definem quais registradores serão acessados, os inteiros, ou ponto flutuante, no terceiro estágio. Após essa divisão do pipeline em dois caminhos no estágio Register Select, entre inteiros

e flutuantes, se tem novamente a divisão, agora em três caminhos possíveis, no quarto estágio que corresponde a execução, sendo eles: execução de inteiros (Exe INT), unidade de ponto flutuante (FPU) e acessos a memória (Exe MEM). Por fim o resultado é escrito no estágio final, Write Back, a Figura 2.29 demonstra a lógica do pipeline.

Para as ocorrências de hazards de dados RAW, se realiza a técnica de forwarding de dados, porém apenas quando a instrução atual, e a anterior tiverem dados do tipo inteiro. Para os hazards de dados WAW se tem a introdução de stalls (ou bubbles) no pipeline, e para as do tipo WAR não se tem ocorrências, pois o núcleo emite instruções em ordem. O processador utiliza de um sistema de predição de branch dinâmico para evitar a perda de desempenho quando se tem a execução de desvios condicionais, no caso de incondicionais, a perda de ciclos de relógio sempre ocorre. O cálculo do endereço alvo de branch realizado no quarto estágio, Execute, logo a perda total é de três ciclos. A unidade de decodificação foi projetada de forma a possuir um nível máximo de lógica com comparadores de 7, 3 e 2 bits.

Figura 2.29 – Visão de topo da arquitetura do processador.



Fonte: Raveendran et al. (2016).

2.5.3 Análise de um Processador Pipeline de 2 a 4 Estágios

Olivieri et al. (2017) exploram micro arquiteturas voltadas a eficiência de consumo de potência em processadores RISC-V, suportando execução de multi-threads de múltiplos kernels computacionais em um único núcleo. O conjunto de instruções utilizado da ISA do RISC-V se concentra na base I, com instruções de 32 bits, e a extensão M no modo privilegiado, vistos como um conjunto mínimo para se executar adequadamente algoritmos

de controle digitais. A análise realizada se focou principalmente na organização do pipeline para a execução das instruções, sendo projetado mais de um caminho de pipeline em termos de profundidade. Para processadores de sistemas embarcados voltados a tarefas de menor carga, o pipeline variou de 2 a 4 estágios, sendo o circuito implementado em dispositivos FPGA da série Xilinx 7.

Cinco núcleos no total foram projetados, todos possuindo as cinco funções: Fetch Instruction (F), Read Registers (R), Decoding (D), Execution (E), Write Back (W). A organização dos diferentes núcleos ficou como na lista abaixo.

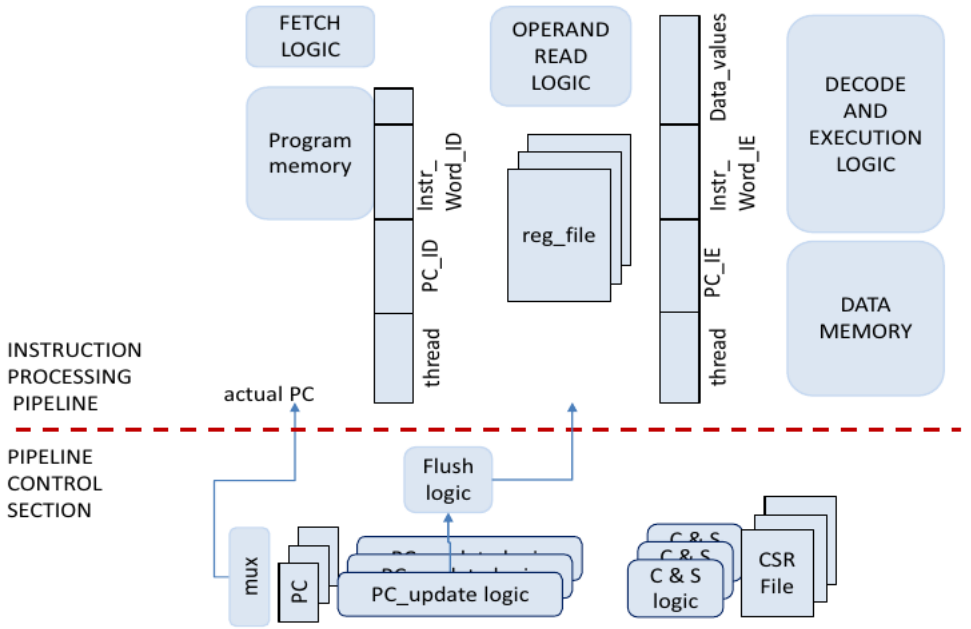
- 2 estágios: F / RDEW;
- 3 estágios: F / R / DEW;
- 3 estágios: F / RD / EW;
- 4 estágios: F / RD / E / W;
- 4 estágios: F / R / DE / W;

Uma representação do pipeline de três estágios "F / RD / DEW"pode ser verificado na Figura 2.30. Com o objetivo para implementações de baixo custo, não se implementou nenhum hardware dedicado para predição de desvios condicionais, logo se tem sempre a perda de performance em branch taken, sendo essa uma das razões para não se aumentar o número de estágios, pois afetaria de forma negativa a perda de ciclos de relógio em instruções de desvio. O mesmo se aplica para as dependências de dados, não se tem o uso de forwarding, sendo sempre parado o pipeline pelo número de ciclos necessários até a escrita do resultado, na função write back (W). Como as perdas se aplicam a instruções de uma mesma thread, os núcleos tendem a melhorar a performance com a execução de múltiplas threads ativas no mesmo pipeline.

2.5.4 Taiga

Foi apresentando por Matthews e Shannon (2017) um processador baseado na ISA do RISC-V, nomeado de Taiga, que executa as instruções das extensões M e A, ou seja, instruções de multiplicação/divisão e operações atômicas, com a base RV32I, sendo o conjunto final chamado de RV32IMA. O seu projeto foi realizado de forma a suportar sistemas de memória compartilhados baseados no Linux, sendo o seu sistema altamente configurável, possibilitando escolhas como o uso de caches, TLBs e operações de multiplicação/divisão. O Hardware foi criado considerando de forma a aumentar a eficiência em implementações nas FPGAs, com uma descrição em SystemVerilog, dando suporte a ambas as FPGAs da Intel e da Xilinx. Os dados de recursos utilizados da FPGA sobre a qual se trabalhou, Xilinx Zynq X7CZ020, foram demonstrados através de uma compara-

Figura 2.30 – Esquemático de uma das configurações de pipeline utilizadas por Olivieri et al. (2017). Consiste em três estágios, Fetch Instruction (F), Read Registers (R), e por fim Decode, Execute and Write Back (DEW).

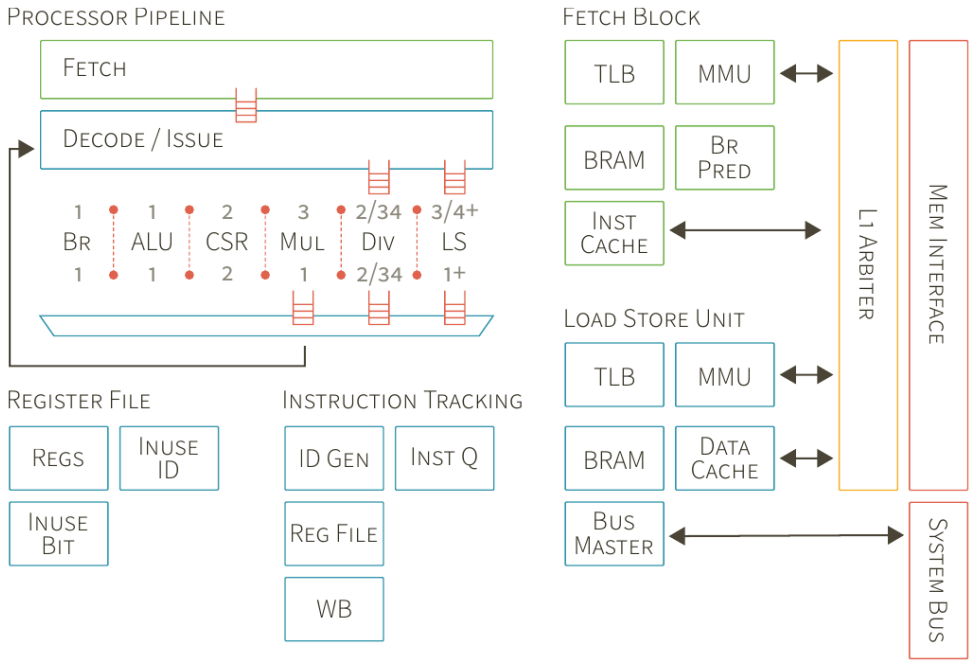


Fonte: Olivieri et al. (2017).

ção de utilização da lógica disponível no CI, assim como a máxima frequência possível de operação para três configurações diferentes, mínima, comparação e completa.

A configuração mínima consiste no núcleo sem multiplicações/divisões, TLBs/MMUs e caches, assim como operações atômicas, a forma completa possui uma cache 4-way de 16KB de dados e 32KB de memória local compartilhada pelas unidades de Fetch e Load/Store. Na configuração de comparação se limitou o núcleo a componentes semelhantes a de outros já existentes com a ISA RISC-V, Rocket e LEON3, para fins de comparação. A faixa de frequência obtida na qual o processador opera é de 103 MHz, partindo da configuração máxima, até 111 MHz na completa, na comparação obteve-se uma frequência de 104 MHz. Na configuração o tamanho do núcleo é próximo de 1,5 vezes o tamanho do obtido na configuração mínima. O sistema também foi projetado para a Xilinx Virtex UltraScale+ XCVU9P, obtendo-se uma frequência de até 254 MHz. A unidade que mais ocupou o circuito é constituída pela unidade de Load/Store, ocupando cerca de 20% dos slices, enquanto que o bloco mais largo é o de decodificação principal, com aproximadamente 17% do hardware. O caminho crítico, durante uso da configuração mínima, está contido na etapa write-back, na forma utilizada para comparação, também esta no write-back, e no circuito completo, o caminho crítico é formado pela data TLB para as tags de dados. A Figura 2.31 contém um esquemático detalhado do processador Taiga.

Figura 2.31 – Detalhes do processador Taiga por Matthews e Shannon (2017), apresentando seus componentes no pipeline. Os números superiores, nas unidades de execução, indicam a latência de execução para a respectiva unidade, em uma dada instrução, enquanto o número inferior é expressão de taxa de transferência como instruções por X ciclos. Uma barra significa que se tem dois possíveis caminhos, e o símbolo de adição, latência mínima.



Fonte: Matthews e Shannon (2017).

2.5.5 ORCA

O núcleo ORCA consiste numa implementação do RISC-V voltada para FPGAs, podendo ser configurada com dois conjuntos de instruções possíveis, RV32I ou RV32IM (VECTORBLOX, 2015). Sua descrição de hardware foi realizada em VHDL, aonde se definiu diversos valores genéricos para controlar o núcleo, que também servem para configurar o hardware, como:

• REGISTER_SIZE = 32, tamanho dos dados contidos nos registradores;

• RESET_VECTOR = 0x0000 0000, endereço da primeira instrução a ser executada;

• INTERRUPT_VECTOR = 0x0000 02000, endereço de desvio para interrupções recebidas;

• BTB_ENTRIES = 0, número de entradas do Branch Target Buffer (BTB);

• MULTIPLY_ENABLE = 0, habilitação do hardware de multiplicação;

• PIPELINE_STAGES = 5, número de estágios do pipeline, podendo escolher entre 4 ou 5;

• FAMILY = GENERIC, habilita certas portabilidades e otimizações para FPGAs de famílias específicas, sendo os valores possíveis "GENERIC", "INTEL", "LATTICE", "MICROSEMI"e "XILINX".

Diversas outras configurações são disponibilizadas para configuração do dispositivo, conforme VectorBlox (2015). O genérico "BTB_ENTRIES", quando colocado com valor 0, elimina o preditor de branch, e para um valor acima, implementa as entradas da BTB como simple dual-port distributed RAMs, com mapeamento direto. No caso de desabilitação do hardware de multiplicação, uma exceção de instrução ilegal vai ser sinalizada ao detectar a execução de qualquer instrução de multiplicação da extensão M do RISC-V. Outra característica a destacar é a possível escolha entre 4 ou 5 estágios de pipeline, supondo que se escolha 4, se elimina um registrador do hardware, salvando área, porém a custo de frequência máxima de operação. ORCA também possui suporte para diferentes interfaces de memória, e da mesma forma que com o núcleo, utilizada de genéricos para configurar algumas de suas características. Suas interrupções utilizam de um controlador simples, não padrão, com dois registradores de controle/status, de 32 bits de largura.

2.5.6 PULPino

O PULPino consiste em um projeto open-source por (ETHZURICH, 2015), formado por um microcontrolador de núcleo único para 32 bits, possuindo a possibilidade de ser configurado para uso de duas variações, núcleo RISCY ou zero-riscy. RISCY é um núcleo com com com emissão e conclusão em ordem, assim como emissão única de instruções, composto por um pipeline de quatro estágios, implementando o conjunto RV32I do RISC-V, extensão C para instruções compressas RV32C, e também instruções de multiplicação e divisão da extensão M, RV32M. Contando ainda com a possibilidade do uso de instruções de ponto flutuante simples, RV32F, caso se deseje configurar dessa forma. O seu projeto teve como foco aumentar a eficiência energética em aplicações de processamento de sinais com baixo consumo de potência, tendo também presente, além das instruções a nível de usuário, as especificações do manual de especificações a nível privilegiado do RISC-V, versão 1.9.

A outra variação do núcleo, zero-riscy, consiste também em um núcleo com in-order, porém com pipeline de apenas dois estágios, oferecendo suporte total a base I e a extensão C, RV32IC, e podendo ser configurado para uso da extensão M. Somado a isso, se opto pela possibilidade de utilizar também a configuração de registradores da base E do RISC-V, tendo como alvo não somente a eficiência energética, mas também de área do dispositivo. Assim como o RISCY, implementa o conjunto de nível privilegiado versão 1.9 do RISC-V.

O PULPino conta com diversos dispositivos E/S para trocas de dados com dispositivos externos, como I2S, I2C, SPI e UART. A plataforma está disponível para simulações RTL, implementações FPGA, na qual pode-se utilizar a placa ZedBoard para síntese. Em janeiro de 2016 foi criado um Application Specific Integrated Circuits (ASIC) do PULPino de 65nm

3 METODOLOGIA E ESPECIFICAÇÕES

Esta seção contém a metodologia com as especificações utilizadas no desenvolvimento do processador, que possuí duas implementações, uma com um núcleo multicíclico, e outro com pipeline. A primeira leitura realizada foi a do manual de instruções do RISC-V (WATERMAN; ASANOVIć, 2017a), a fim de se familiarizar com a ISA, e determinar as especificações iniciais do projeto. O mesmo oferece instruções de nível de usuário, sendo separado em outro manual as de nível privilegiado, que ainda não estão finalizadas e não são consideradas padrões do RISC-V (WATERMAN; ASANOVIć, 2017b), optando-se por não implementa-las.

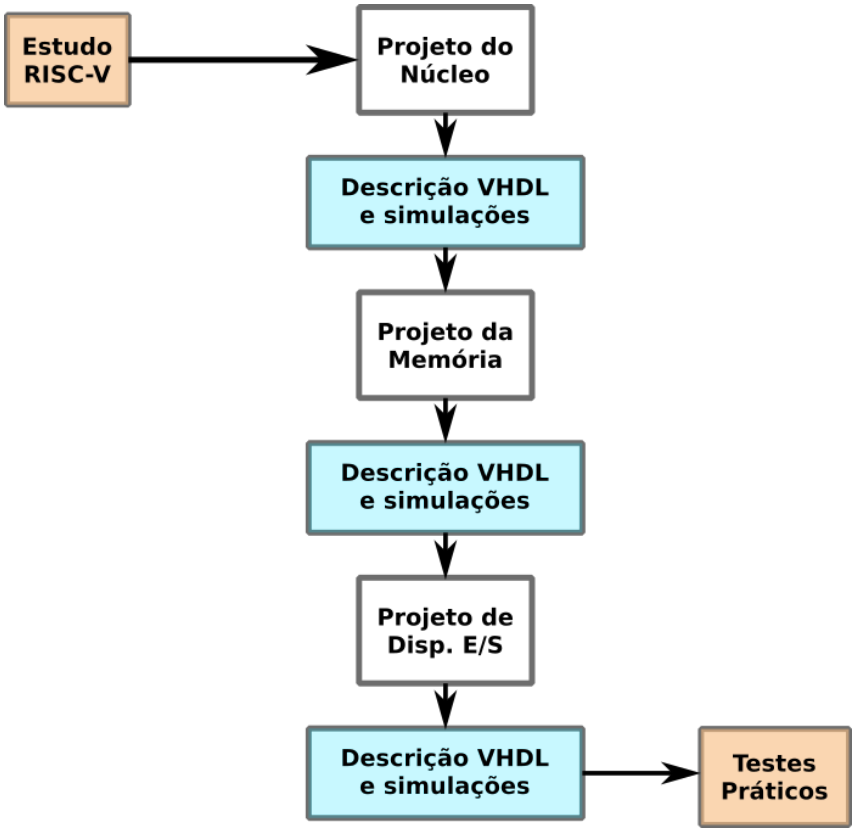
Após a revisão da ISA alvo, se divide em duas etapas o projeto, uma para o processador com núcleo multicíclico, e outra para o pipeline, conforme o diagrama da Figura 3.1. Em ambas as etapas o primeiro passo era o estudo do caminho de dados apresentado por Patterson e Hennessy (2004), do qual modificou-se para se adaptar a características presentes na ISA alvo. O núcleo multicíclico consiste em um projeto mais experimental, pois segundo Waterman e Asanović (2017a, p. 68), não recomenda-se o uso da extensão C do RISC-V (ISA alvo do núcleo) sem nenhuma de suas quatro bases. Também se utilizou como base para revisão de circuitos digitais a obra de Tocci, Widmer e Moss (2011).

Após finalizado o projeto do núcleo multicíclico, a atenção é voltada aos demais componentes do processador, necessários para execução de programas, sendo eles os componentes de memória, e dispositivos E/S. Durante muitos momentos do projeto é preferível o uso de memórias genéricas, com fácil implementação, e uso rápido tanto em simulações quanto testes práticos. Sendo somente nas etapas finais do projeto trocada por uma mais complexa, com maior capacidade de armazenamento.

Para testar o processador na prática, é necessário uma forma de comunicação, a fim de averiguar o que ocorre dentro do núcleo. Uma unidade de debug seria o ideal, mas como o foco do processador se dá no núcleo, buscou-se outros componentes. Utilizou-se como referência o estudo de uma UART, demonstrado por Hexsel (2012, p. 225), para realizar a comunicação serial entre o processador e um computador pessoal. Também se criou um contador de 32 bits, para servir como periférico temporizador do processador.

Finalizado as três etapas, núcleo, memória, e dispositivos E/S, se executou o teste das instruções do hardware implementado na FPGA Spartan-6 para o núcleo multicíclico. Comprovando seu funcionamento, realizou-se novamente o ciclo, porém agora com foco em um núcleo pipeline. As demais unidades (memória e dispositivos E/S) são reaproveitadas, e alteradas o suficiente para funcionar com o novo núcleo. Durante toda a descrição VHDL, se utilizou como referência o livro texto de Pedroni (2010), assim como manuais tanto da FPGA alvo, quanto da placa de desenvolvimento, nexys3 (XILINX, 2011b; XILINX, 2006; XILINX, 2013; XILINX, 2011a; XILINX, 2009a; XILINX, 2009b).

Figura 3.1 – Fluxo de projeto.



Fonte: Autor.

3.1 INSTRUÇÕES DO RISC-V

Para começar o projeto foi necessário definir-se inicialmente quais instruções seriam executadas dos conjuntos escolhidos como especificação inicial, base E e extensão C do RISC-V. Da extensão C, foram selecionadas todas as instruções com exceção das que envolviam qualquer tipo de operação com dados do tipo não inteiro, ou de 64 bits, pois o processador irá operar sobre dados de 32 bits. Também se excluiu instruções de controle, como "c.EBREAK", pois nessa etapa inicial não se tem por objetivo o uso de nenhum sistema operacional para retorno dos programas. Da base E, manteve-se a mesma escolha, sem instruções que envolviam chamadas ao sistema, ou então para uso de unidades de debug, a Tabela 3.1 contém todas as instruções implementadas. Lembrando que o núcleo multicíclico será capaz de executar apenas o conjunto C, enquanto o pipeline todas as instruções selecionadas. Para conferência, está contido na Tabela 3.2 as instruções do conjunto RV32EC que foram descartadas para o projeto.

Totalizou-se 25 instruções compressas da extensão C, e 37 da base E do RISC-V a serem utilizadas. É possível realizar tanto acessos alinhados quanto desalinhados, que apesar de o RISC-V suportar não o recomenda, pois pode ocorrer de executar com uma performance muito inferior (WATERMAN; ASANOVIć, 2017a, pág. 19). Logo optou-se por

Tabela 3.1 – Tabela de instruções selecionadas para implementação da base E e extensão C do RISC-V.

Extensão C	Base E
c.addi4spn	lui
c.lw	auipc
c.sw	jal
c.nop	jalr
c.addi	beq
c.jal	bne
c.li	blt
c.addi16sp	bge
c.lui	bltu
c.srli	bgeu
c.andi	lb
c.sub	lh
c.xor	lw
c.or	lbu
c.and	lhu
c.j	sb
c.beqz	sh
c.bnez	sw
c.slli	addi
c.lwsp	stli
c.jr	sltiu
c.mv	xori
c.jalr	ori
c.add	andi
c.swsp	slli
—	srli
—	srai
—	add
—	sub
—	sll
—	slt
—	sltu
—	xor
—	srl
—	sra
—	or
—	and

Fonte: Próprio Autor.

manter-se apenas o uso de acessos alinhados. Com as instruções já definidas, obtém-se então algumas especificações iniciais de projeto, a respeito do hardware e seu comportamento:

Tabela 3.2 – Tabela de instruções não selecionadas para implementação da base E e extensão C do RISC-V.

Extensão C	Base E
c.fld	fence
c.lq	fence.i
c.flw	ecall
c.ld	ebreak
c.fds	csrrw
c.sq	csrrs
c.fsw	csrrc
c.sd	csrrwi
c.addiw	csrrsi
c.srli64	csrrci
c.srai64	—
c.subw	—
c.addw	—
c.slli64	—
c.fldsp	—
c.lqsp	—
c.flwsp	—
c.ldsp	—
c.ebreak	—
c.fsdsp	—
c.sqsp	—
c.fswsp	—
c.sdsp	—

Fonte: Próprio Autor.

- Processador de 32 bits (base 32E);
- Banco de registradores possui 16 registradores de uso geral;
- Operações unicamente sobre dados do tipo inteiro.
- Acessos alinhados a memória apenas.

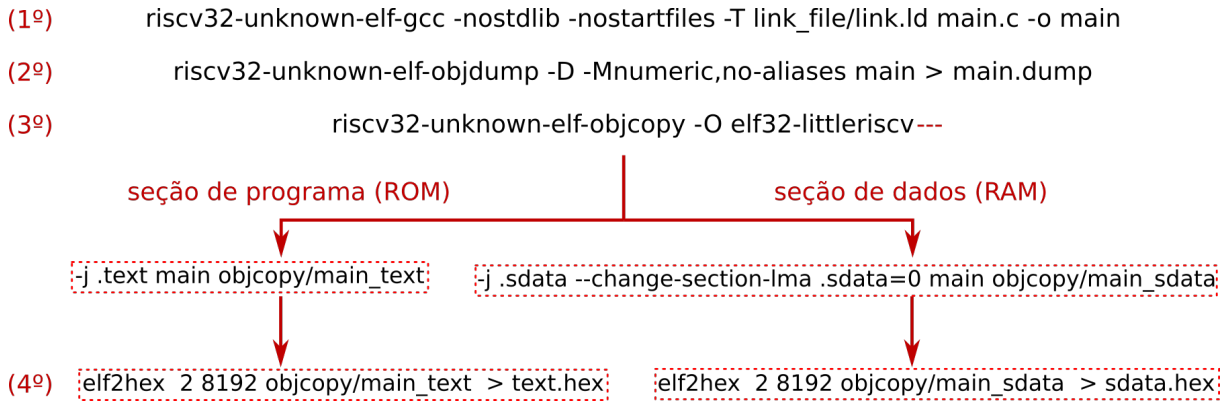
3.1.1 Compilador

O RISC-V conta com diversas ferramentas para desenvolvimento de softwares, incluindo linguagens como C ou C++, e também para manipulação de códigos compilados, entre outras finalidades. Duas linguagens são desejadas no projeto, ASM e C, sendo no caso do multicíclico o uso de assembly uma necessidade, pois o compilador não pode gerar códigos apenas com instruções pertencentes a extensões (sem base). Na versão 8.2 do RISC-V GCC o suporte a base E ainda não é pleno, ao compilar e instalar o conjunto de ferramentas para desenvolvimento de softwares se gera erros no processo de escritas

escolhendo a arquitetura desejada para o projeto. Logo é necessário que todo o conjunto de ferramentas a ser utilizado seja compilado e instalado para a arquitetura RV32EC em específico. Dessa forma se passa os argumentos que definem a arquitetura apenas na instalação do GCC, se tornando a opção alvo padrão, e oferecendo o suporte mínimo necessário.

Durante a compilação de programas, procurou-se utilizar as opções que levassem a geração de um código sem suporte a SO, ou seja, com ausência de chamadas ao sistema, por exemplo. O processo completo para a geração do programa está apresentado na Figura 3.2, aonde o primeiro comando consiste na compilação do código C/ASM, para o executável. Dentre os argumentos possíveis de passar para o compilador, se destaca o responsável pela indicação do uso de um script de link externo, e também os argumentos que definem a arquitetura alvo que irá executar o programa, "march"e "mabi". O primeiro, "march", define as instruções a serem utilizadas, sendo obrigatório o uso de pelo menos uma das quatro bases, deixando as extensões a escolha do programador, que no projeto definiu-se como -march=rv32ec".

Figura 3.2 – Processo de geração do programa a ser carregado para a FPGA. O primeiro passo corresponde ao comando de compilação. Após isso se utiliza o comando objdump para verificação do programa gerado, sendo então separada suas seções de programa e dados pela ação do objcopy. Por fim executa-se o software elf2hex para extrair os arquivos hexadecimais de cada seção.



Fonte: Autor.

Por fim o segundo argumento, "mabi", serve para definir a ABI utilizada, determinando os tipos de dados entre inteiros ou ponto flutuante, sendo então essa decisão uma dependência de o primeiro conter na arquitetura declarada instruções que realizam esse tipo de operação. A ABI utilizada foi "ilp32e", indicando dados do tipo inteiro padrão de 32 bits, e adaptados para funcionar com a base E conforme indicado no último caractere. Como as ferramentas foram compiladas para utilizar como alvo padrão a arquitetura desejada, esses argumentos não precisam ser passados no momento de execução.

Para verificar o código obtido em ASM na compilação, se utiliza o comando "obj-

dump", de forma a não demonstrar pseudo-instruções e também os valores de x0 a x15 dos registradores de acordo com as convenções, como SP (stack pointer - x2) ou zero (x0), por exemplo. Para carregar o programa nos blocos de RAM, é necessário a separação das seções de programa e dados, através do comando "objcopy", no qual é indicado o formato alvo de saída, no caso elf32-littleriscv. Executa-se o comando duas vezes, uma para extrair a seção de texto, e outra para extrair a seção de dados, sendo na segunda necessário ainda alterar o endereço LMA (load address memory) inicial da seção para 0.

Tanto o comando de compilação, quanto o objdump e o objcopy são instalados em conjunto com as ferramentas RISC-V GNU Toolchain, porém para converter cada arquivo de seção já separados pela ação do objcopy, para um formato hexadecimal, se utiliza do software "elf2hex". Este é instalado com o RISC-V Frontend Server, e para executa-lo se especifica um arquivo de entrada, largura em número de bytes, e profundidade. Executando uma vez sobre cada seção, se obtém os hexadecimais a serem carregados para a memória ROM e RAM, na FPGA. Vale ressaltar que o formato hexadecimal consiste em puro texto, com apenas as instruções, e não no formato hexadecimal da intel, a saída do programa elf2hex de um programa genérico com uma única instrução de 32 bits seria:

```
04b5
018b
```

3.1.2 Teste das Instruções

Para testar se as instruções estavam sendo executadas corretamente, se utilizou de um programa que opera sobre todos os registradores de uso geral (x0-x15) para cada instrução, incluindo o x0 (constante), para verificar que o mesmo é ignorado em escritas. Para a instrução ADDI da base E, utilizou-se de um somatório:

```
addi_test:
    addi    x0,x0,7
    addi    x1,x0,-7
    addi    x2,x0,14
    addi    x3,x0,-14
    addi    x4,x0,28
    addi    x5,x0,-28
    addi    x6,x0,56
    addi    x7,x0,-56
    addi    x8,x7,8
    addi    x9,x8,8
    addi    x10,x9,8
```

```
    addi    x11,x10,8
    addi    x12,x11,8
    addi    x13,x12,8
    addi    x14,x13,8   # x14  = 0
    addi    x15,x14,1   ## if (x15 = 1) -> ADDI -> OK!

addi_uart:
    add     x13,x0,x15
    jal     x1,uart_write_loop
```

que no final deve resultar em 1 decimal (em x15). Para uso na prática se utiliza então da UART para enviar o resultado ao computador pessoal, e comprovar o funcionamento do hardware. Isso causa uma dependência de que para testar qualquer instrução, se tenha a rotina para envio de dados pela UART em perfeito funcionamento, incluindo a instrução de desvio JAL. O código de teste completo para as instruções da base E e extensão C podem ser visualizados nos Apêndices A.2 e A.3, respectivamente.

3.2 NÚCLEO

O projeto do caminho de dados de ambos os núcleos do trabalho, um multicíclico e outro fazendo uso da técnica de pipeline, seguiram o modelo inicial utilizado por Patterson e Hennessy (2004), acrescentando as características extras necessárias a ISA alvo, RISC-V. Para o caminho de dados multicíclico utilizou-se somente da extensão C do RISC-V, provendo instruções compressas de 16 bits, porém, trabalhando sobre dados de 32 bits de comprimento.

Já para o núcleo com pipeline se almejou o conjunto RV32EC, base E (instruções de 32 bits) com a extensão C. Porém, antes de executar uma instrução compressa no núcleo do pipeline, ele a decodifica e expande para sua respectiva de 32 bits, no estágio de busca de instrução. Isso diminui o requerimento de hardware presente no pipeline para se adaptar a tantas instruções possíveis de execução, e não acrescenta tanta complexidade a decodificação. É referenciado o núcleo multicíclico de RVC, e ao pipeline de RV32EC, que indicam seus conjuntos de instruções implementados.

O barramento para troca de informações com a memória é único e bidirecional, assim como o de endereços de acesso também é único, tanto para instrução quanto para dados. Em ambas as abordagens de núcleo se tem componentes comuns a sua execução a serem identificados: contador de programas, banco de registradores, unidade lógica e aritmética (ALU), unidade de controle/decodificação, e por fim uma unidade de decodificação do campo imediato, que por característica do RISC-V, está embaralhado nos campos

da instrução.

Apesar de os componentes em comum realizarem a mesma função em ambos os núcleos, seu funcionamento difere significativamente devido ao modelo do núcleo (multicíclico e pipeline), tornando-se necessária a separação por núcleo, não generalizando totalmente o comportamento dos mesmos para ambos os projetos do caminho de dados. Um exemplo significativo é a unidade de controle, que no multicíclico é implementada como uma máquina de estados, e no pipeline como um elemento essencialmente combinacional. Como consequência do uso da base E, e a redução do número de registradores de uso geral de 32 para 16, ignora-se o MSB do campo das instruções que contém endereços do banco, pois o mesmo não acrescenta nenhuma informação. Apesar de as instruções da base E não serem utilizadas no núcleo RVC, se tem presente sua característica de redução do tamanho do banco de registradores.

3.2.1 Núcleo Multicíclico - RVC

O caminho de dados multicíclico está representado na Figura 3.3, com os diversos componentes necessários a sua execução, assim como os sinais de controle para seu correto funcionamento. Alguns registradores temporários são alocados internamente para armazenar valores a serem utilizados em ciclos posteriores de execução, como característica dos caminhos de dados com abordagem multicíclica de execução, sendo eles os registradores de saída de dados do banco de registradores (RegA e RegB), registrador de saída da unidade lógica e aritmética (RegALU), registrador de armazenamento da instrução (Reg. Instr) e registrador de dados da memória (Reg. de Dados). O registrador de instrução possui sua escrita habilitada separadamente dos demais, pelo sinal de controle "Instr_W", pois sua saída deve se manter estável pelos múltiplos ciclos de execução da instrução, a fim de não alterar o comportamento da unidade de decodificação. Outro registrador fundamental é o contador de programa (PC), possuindo uma porta de entrada de 32 bits, e uma de saída de mesmo tamanho, apontando para a instrução a ser acessada da memória, sendo sua escrita habilitada pela lógica do sinal de controle "PC_W" com o resultado do teste de instruções branch, indicado pelos sinais "zero" da ALU e o de controle "PC_W_cond". Multiplexadores estão presentes no caminho de dados a fim de controlar o fluxo de dados de alguns componentes, suas funções são:

• MUX 1: define o endereço de acesso a memória entre a saída do PC, e a saída do RegALU, sendo controlado pelo sinal "IouD";

• MUX 2: define o dado a ser escrito no banco de registradores entre a saída do Reg. de Dados, RegALU, o valor de saída de PC (utilizado em instruções jump and link, JAL), ou então do imediato de sinal estendido (conexão utilizada unicamente para instruções c.LUI),

sendo controlado pelo sinal "MemToReg";

• MUX 3: define o endereço de escrita do banco de registradores entre o mesmo utilizado para leitura da porta A ou B do banco, ou então pela constante 1, pois o registrador x1 corresponde ao link register na ABI do RISC-V (logo é utilizado em instruções jump and link, JAL), sendo controlado pelo sinal "RegDstW";

• MUX 4: define o operando A da ALU entre o valor de PC (para seu incremento, ou então cálculo de end. de desvio), saída de RegA, ou então a constante 0, que somada com o operando B acaba por apenas copiar o dado, resultando na instrução c.MV, sendo controlado pelo sinal "ALUoprA";

• MUX 5: define o operando B da ALU entre o valor de RegB, a constante 2 para incremento do PC, o sinal imediato estendido pelo MSB ou o sinal imediato estendido com 0, sendo controlado pelo sinal "ALUoprB";

• MUX 6: define o endereço de entrada de PC entre a saída da ALU, ou de RegALU, sendo controlado pelo sinal "PC_Src";

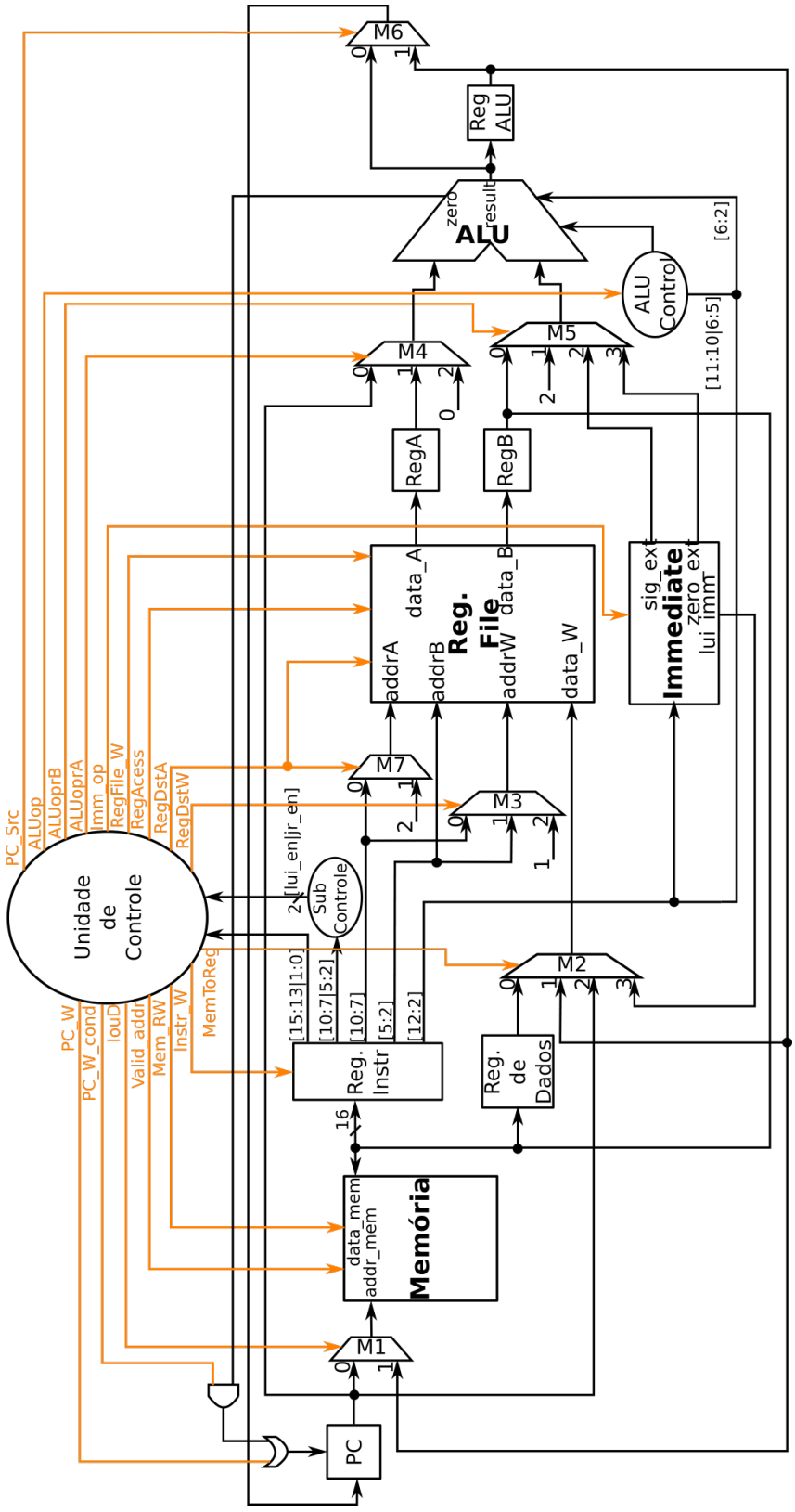
• MUX 7: define o endereço de leitura da porta A do banco de registradores entre o indicado pelo campo [10:7] da instrução, ou então a constante 2 que é usado como stack pointer na ABI do RISC-V, controlado pelo sinal "RegDstA".

Na Figura 3.4 se visualiza as portas de entrada e saída do núcleo. Os sinais de controle de entrada são o sinal de relógio (clk), cujo o qual o núcleo opera na borda de subida, o sinal de reset (rst) ativo em nível 1, e um sinal de espera por dados da memória síncrona (wait_mem), que ao estar ativo em nível 1 trava o núcleo. Sua saída de controle "we_l" corresponde a habilitação de escrita da memória (sinal "Mem_RW" do caminho de dados), em nível 0 indica a operação de escrita, e nível 1 de leitura. O barramento de dados é bidirecional e de 32 bits, possuindo também um barramento de saída de endereço de acesso, e um sinal de validade do endereço, para evitar acessos indesejados e consequentes execuções de instruções que não deveriam ocorrer.

3.2.1.1 Banco de Registradores

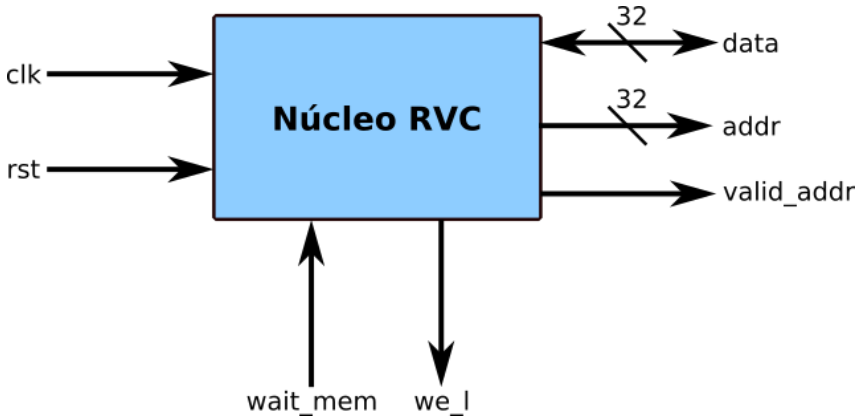
Apesar de o núcleo multicíclico implementar apenas o hardware necessário para a execução das instruções da extensão C, utilizou-se da característica ditada pela base E relacionada ao banco de registradores, visto que existe compatibilidade entre a base E e extensão C do RISC-V. Sendo assim, o banco de registradores é de tamanho 16x32, 16 registradores de 32 bits, possuindo duas portas de entrada para endereço de leitura, addrA e addrB, ambas de 4 bits, e uma para o endereço de escrita de mesmo tamanho, addrW, assim como um sinal de habilitação da ação de escrita, "RegFile_W". A Figura 3.5 apresenta as portas de controle do banco de registradores, assim como as de entrada e

Figura 3.3 – Caminho de dados multicíclico.



Fonte: Autor.

Figura 3.4 – Visão de topo do núcleo RVC, evidenciando suas portas de entrada e saída.



Fonte: Autor.

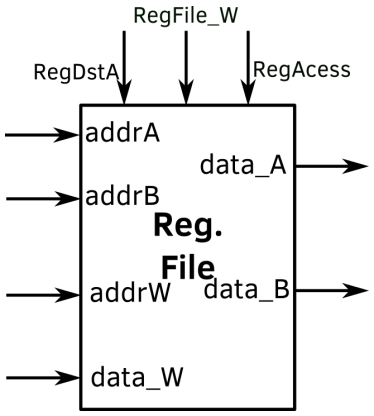
saída.

Duas portas de leitura de dados correspondentes aos endereços de leitura, com largura de 32 bits, e uma porta de entrada para de dado a ser escrito no endereço addrW. Na ISA do RISC-V o registrador x0 é uma constante 0, logo sua escrita é ignorada nesse endereço, sendo utilizado seu valor em diversas pseudo-instruções como load immediate. O conjunto RVC, para obter diversas de suas instruções compressas, limitou o acesso ao banco de registradores aos de rótulo x8 até x15 em certas execuções, dessa forma é necessário um sinal de controle para indicar quando esse acesso é limitado, nomeado de "RegAcess", que realiza uma operação OR com o MSB do endereço de leitura/escrita do banco, o colocando em nível 1 quando necessário. Outro registrador nomeado pela ABI do RISC-V é o x2 como stack pointer, que em instruções específicas deve ser utilizado como leitura na porta A do banco. Para evitar que o sinal "RegAcess"acidentalmente coloque me nível 1 seu MSB, o que levaria seu valor de 2 para 10, outro sinal é necessário para indicar que a lógica de "RegAcess"deve ser ignorada em determinados casos, sendo esse sinal de controle rotulado de "RegDstA".

3.2.1.2 Unidade Imediato

O imediato utilizado em diversas instruções deve esperar o sinal de controle da unidade de decodificação para ter seu valor determinado, pois seus bits estão embaralhados ao longo do formato da instrução. O bloco responsável por isso é composto de multiplexadores que selecionam para saída dois sinais imediato, um estendido com 0, e outro com o MSB, sendo o campo opcode da instrução que determina qual será utilizado. Internamente o bloco decodifica as duas saídas de sinal imediato com base nos tipos de instruções que as utilizam, essa separação facilita a implementação da multiplexação por bit separada-

Figura 3.5 – Bloco lógico do banco de registradores.



Fonte: Autor.

mente, sendo esse o objetivo primário do embaralhamento do seu sinal, diminuir o custo de hardware por multiplexadores.

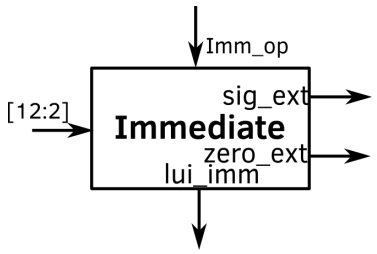
Esta contido na Tabela 3.3 a divisão dos imediatos a serem multiplexados (decodificados) para cada uma das duas saídas, estendida com 0 ou o MSB. A única instrução que envolve o sinal imediato que ficou de fora da decodificação foi c.LUI (load upper immediate), pois ela atribui seus campos para a posição [12] e superiores, enquanto todas as outras possuem apenas campos correspondentes aos bits [11-0]. Por conta dessa diferença, o sinal imediato de c.LUI é conectado diretamente a saída do registrador de instrução, a fim de não aumentar a unidade de decodificação do imediato. A Figura 3.6 demonstra o bloco lógico da unidade imediato, aonde o sinal de controle, "Imm_op", possui 4 bit de comprimento, sendo os 2 MSB responsáveis pelos multiplexadores internos do sinal estendido com 0, e os 2 LSB pelo estendido pelo MSB, que está sempre posicionado no bit 12 da instrução. O campo [12:2] da instrução é a entrada da unidade que possui todos os bits possíveis de conter imediato.

Tabela 3.3 – Divisão de decodificação do campo imediato, embaralhado ao longo das instruções RVC. Instruções agrupadas possuem o sinal imediato decodificado da mesma forma, outros refere-se a todos os demais com sinal estendido, utilizados em instruções ANDI, ADDI, LI, entre outras, todas com campo do sinal imediato igual.

Estendido com 0	Estendido com MSB
ADDI4SPN	ADDI16SP
LW / SW	J / JAL
LWSP	BEQZ / BNEZ
SWSP	–outros–

Fonte: Autor.

Figura 3.6 – Bloco lógico da unidade decodificadora do campo imediato da instrução.



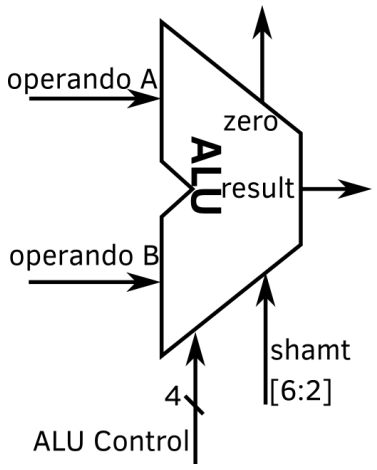
Fonte: Autor.

3.2.1.3 Unidade Lógica e Aritmética

A unidade lógica e aritmética possuí duas portas de entrada, uma para cada operando, e uma saída, todas de 32 bits, assim como uma entrada de controle de 4 bits que define sua operação, ALU control, e um sinal de comparação a zero do resultado. O sinal de comparação a zero resulta em nível lógico 1 quando o operando A é igual a 0 em qualquer instrução em execução, com exceção da instrução c.BNEZ (branch if not equal to zero), em que se nega o sinal. Também se tem presente uma porta de entrada nomeada de shamt (shift amount), que define o tamanho do deslocamento em instruções de shift sobre o operando A da ALU, sendo seu comprimento de 5 bits, e correspondente ao campo [6:2] da instrução.

A Figura 3.7 apresenta o bloco lógico do PC e da ALU, com suas portas de entrada e saída. Todas as execuções lógicas/aritméticas da ALU são realizadas em um único ciclo de relógio. A Tabela 3.4 contém a codificação das suas operações possíveis de acordo com seu sinal de controle.

Figura 3.7 – Bloco lógico da ALU.



Fonte: Autor.

Tabela 3.4 – Codificação das operações da ALU. O sinal ALU Zero corresponde ao sinal de saída do comparador utilizado para instruções de branch, sendo o sinal negado quando o MSB de ALU control possuí nível lógico 1, indicando Branch Not Equal to Zero.

ALU control	Operação	ALU Zero
0000	ADD/BEQZ	Zero
0001	SUB	Zero
0010	OR	Zero
0011	AND	Zero
0100	XOR	Zero
0101	SLL	Zero
0110	SRL	Zero
0111	SRA	Zero
1xxx	ADD/BNEZ	NOT(Zero)

Fonte: Autor.

3.2.1.4 Unidade de Controle

A unidade de controle sendo constituída essencialmente por uma máquina de estados possibilitava duas abordagens direta sobre seu projeto, sendo o de implementa-la como uma máquina de mealy ou de moore. Diferente da lógica de execução do núcleo multicíclico exemplificado por Patterson e Hennessy (2004), onde se realiza a leitura do banco de registradores em paralelo a decodificação da instrução, neste projeto se tem a necessidade da decodificação com antecedência. Isso se deve ao fato de algumas instruções da extensão C do RISC-V acessarem somente os registradores de x8 a x15 do banco de registradores, conforme descrito na Seção 3.2.1.1.

A fim de se obter então uma resposta mais rápida da decodificação se utilizou de uma máquina de mealy. Os campos da instrução utilizados para decodificação são os correspondentes aos bits [15:13] e [1:0], chamados de funct3 e opcode, respectivamente. A máquina de estados se comporta conforme o diagrama da Figura 3.8, aonde quando não se especifica o estado de um sinal, ele não importa, com exceção dos de escrita ou acesso a memória que estão obrigatoriamente inativos (nível 0). Sinais que aparecem, porém sem um estado definido, indicam que são relevantes no estado atual, porém seu valor pode variar de acordo com a instrução, como no estado J por exemplo, em que enquanto a instrução c.j (jump) não escreve no banco de registradores, a c.jal (jump and link) o faz. Alguns estados são pontos de convergência por realizar tarefas comuns a diversas instruções, sendo o A responsável pela busca da instrução, e também o primeiro a ser executado após o sinal de reset do núcleo, o estado B de decodificação da instrução e leitura do banco de registradores, e o estado D responsável pela operação de escrita no banco de registradores.

O sinal de saída "Valid_addr" da unidade de controle foi ocultado do diagrama de estados, estando ativo em todos os estados de acesso a memória, A, H e I, em nível 1. Ao

utilizar uma memória síncrona se tem presente a necessidade de travar os registradores essenciais ao funcionamento do núcleo a espera do envio do dado, sendo eles todos os registradores temporários, incluindo a máquina de estados. Para evitar que o núcleo fique constantemente recebendo (e agindo sobre) o sinal de espera pela memória, desabilita-se o sinal "Valid_addr". Garantindo assim o funcionamento do núcleo sem interrupções constantes, ao indicar ao decodificador da memória que não se deve realizar um acesso a memória.

Os sinais de saída da unidade de controle possuem as seguintes finalidades:

• PC_W: habilita a escrita do registrador PC em nível lógico 1, sendo utilizado no incremento do mesmo, ou então em desvios incondicionais;

• PC_W_cond: habilita a escrita do registrador PC de acordo com o resultado da instrução de desvio condicional executada, em nível lógico 1;

• IouD: seleciona o endereço de acesso a memória entre o valor atual de PC para nível lógico 0, que é utilizado para buscar instruções, ou então entre a saída de RegALU em nível lógico 1, para acessar dados da memória;

• Valid_addr: utilizado para indicar quando um endereço no barramento é válido para acesso a memória em nível 1;

• Mem_RW: quando em nível 1 indica uma operação de leitura a memória, e em nível 0, de escrita;

• Instr_W: habilita a escrita no registrador temporário de instrução em nível 1;

• MemToReg: seleciona o dado a ser escrito no banco de registradores entre a saída do Reg. de Dados (valor 00), a saída de RegALU (valor 01), a saída de PC (valor 10) ou o sinal imediato para instruções c.LUI (valor 11);

• PC_Src: seleciona valor do endereço a ser escrito no PC, entre a saída da ALUou a saída de RegALU, para os níveis 0 e 1, respectivamente;

• Imm_op: decodifica o campo imediato, o sinal possui comprimento de 4 bits, sendo os 2 MSB utilizado internamente na unidade de decodificação do imediato para o sinal estendido por 0, e os 2 LSB para o estendido pelo MSB;

• ALUop: sinal conectado a subunidade de controle da ALU para definir a sua operação a ser executada, comporta-se de acordo com a Tabela 3.5;

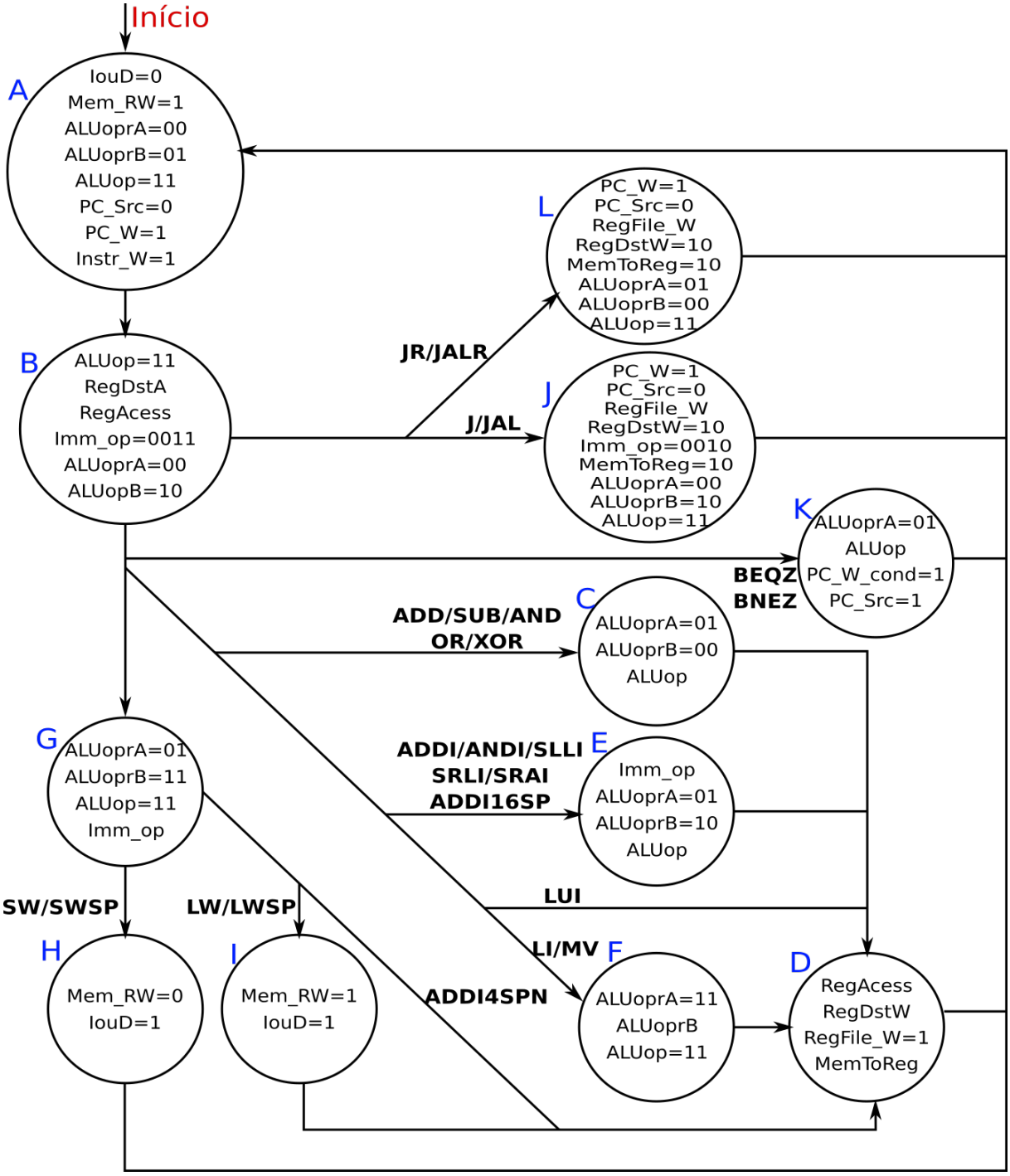
• ALUoprA: seleciona o operando A da ALU entre a saída de PC (valor 00), a saída de RegA (valor 01) ou a constante 0 para copiar registradores por meio da instrução c.MV (valor 10 ou 11);

• ALUoprB: seleciona o operando B da ALU entre a saída de RegB (valor 00), a constante 2 para incremento de PC (valor 01), o sinal imediato estendido pelo MSB (valor 10) ou o sinal imediato estendido com 0 (valor 11);

• RegFile_W: habilita a escrita no banco de registradores em nível lógico 1;

• RegAcess: fixa em nível lógico 1 o MSB dos endereços de leitura das portas A e

Figura 3.8 – Diagrama de estados da unidade de controle, indicando quais instruções são executadas em cada estado. Quando um sinal não está especificado, o mesmo não importa, com exceção de acessos a memória ou escrita, que estão inativos, sinais que aparecem porém com estado indefinido variam com a instrução em execução.



Fonte: Autor.

B do banco de registradores, quando em nível 1, limitando o acesso aos registradores de x8 a x15;

• RegDstA: seleciona o endereço de leitura da porta A do banco de registradores entre o contido no campo [10:7] da instrução para o nível 0, ou a constante 2 (stack pointer na ABI do RISC-V) em nível 1, também evita conflitos com a lógica de RegAcess internamente no banco;

• RegDstW: seleciona o endereço de escrita do banco de registradores entre o da porta A (valor 00), porta B (01) ou a constante 1 para instruções jump and link (valor 10 ou 11).

Tabela 3.5 – Codificação do sinal de controle ALUop. Quando seu valor é 00, a operação é definida pelos campos funct2 e funct1 da instrução, que correspondem aos bits [11:10] e [6:5], respectivamente, sendo que o campo funct2 tem prioridade sobre o funct1. Valores não especificados resultam na operação de adição.

ALUop	ALU Control	Operação
00	—-	—-
01	0101	SLL
10	1000	ADD / BNEZ
11	0000	ADD / BEQZ
funct2:funct1	ALU Control	Operação
00:xx	0110	SRL
01:xx	0111	SRA
10:xx	0011	AND
11:00	0001	SUB
11:01	0100	XOR
11:10	0010	OR
11:11	0011	AND

Fonte: Autor.

Algumas instruções compartilham o mesmo valor para os campos funct3 e opcode, sendo necessário o auxílio de uma subunidade de controle para determinar a instrução em execução, ao verificar outros campos não conectados a unidade principal de decodificação da instrução. A Tabela 3.6 demonstra o conflito do controle em algumas instruções, com o valor de funct3 e opcode, e também o sinal que a subunidade de controle envia para a principal, para resolução de conflitos. A subunidade determina o valor de "jr_en" por meio dos bits [5:2] da instrução, que estão associados ao registrador de leitura da porta B do banco de registradores. Se ele possuir valor nulo, a subunidade envia em nível alto o sinal "jr_en", indicando que é uma instrução do tipo jump register. Para nível 0, é a instrução c.MV ou c.ADD, e em ambos os casos em que se sobrou duas instruções para determinar qual é a correta, o bit [12] da instrução é responsável por essa função. De forma semelhante, para as instruções c.ADDI16SP e c.LUI, se tem que na execução de

c.ADDI16SP o registrador de leitura da porta A do banco é a constante 2 (stack pointer register). Assim, se a comparação der negativa a essa constante, o sinal "lui_en" em nível 1 sinaliza que para a unidade principal de decodificação que é a instrução c.LUI. Com o uso da base E e a consequente redução de 32 para 16 registradores de uso geral ignora-se o MSB do campo que contém o endereço de acesso das portas do banco, pois o mesmo não acrescenta nenhuma informação a decodificação.

Tabela 3.6 – Conflito de campos funct3 e opcode entre instruções, necessitando de auxílio dos sinais providos pela subunidade de controle a fim de resolver a decodificação.

Instruções	[funct3|opcode]	Sinal de Controle
c.ADDI16SP / c.LUI	011|01	lui_en
c.JR / c.MV / c.JALR / c.ADD	100|10	jr_en

Fonte: Autor.

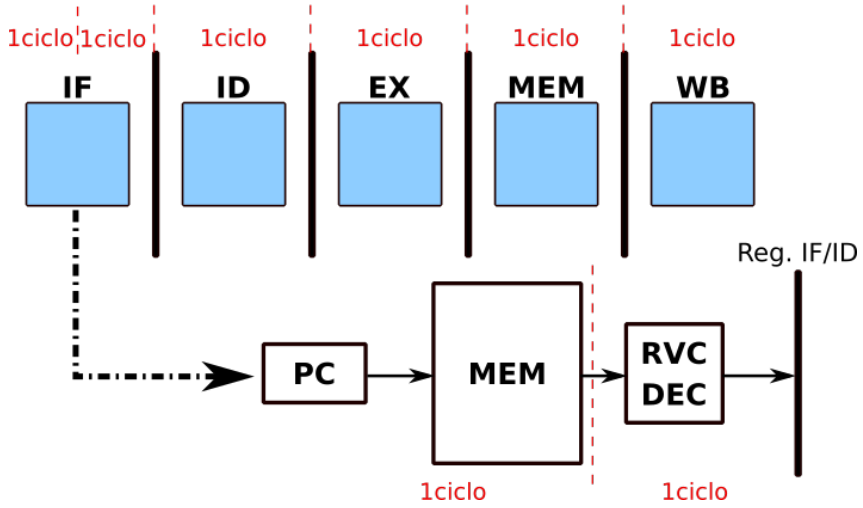
3.2.2 Núcleo Pipeline - RV32EC

Projetado a partir do multicíclico, se modificou o necessário para incluir os registradores de pipeline entre os estágios de execução, sendo seguido o modelo clássico de 5 estágios apresentado por Patterson e Hennessy (2004): Instruction Fetch (IF), Instruction Decode (ID), execute (EX), Memory (MEM) e Write Back (WB). A alteração mais significativa se da por conta do surgimento de uma etapa extra para decodificação da instrução, isso se deve pelo fato da unidade de decodificação, presente no segundo estágio do pipeline, aceitar somente instruções de 32 bits, ou seja, da base E. Assim se torna necessário que se converta as instruções de 16 bits da extensão C que forem buscadas da memória no estágio um, para a correspondente de 32 bits.

A verificação do tamanho da instrução é rapidamente realizada pelos dois LSB, caso um seja diferente de nível lógico 1, corresponde a extensão C, e a mesma deve ser decodificada. O decodificador dedicado a instruções RVC está posicionado, em ordem de execução, no estágio um do pipeline, mas o hardware em si está na saída da memória de programa. A Figura 3.9 demonstra os estágios de pipeline do núcleo, e também destaca o posicionamento do decodificador RVC, "RVC DEC", no estágio IF, também se indica o número de ciclos de execução total do pipeline.

Com o uso de uma memória síncrona, estabelecido pelo uso de blocos de RAM para compor tanto a memória de programa quanto de dados, se tem que após a latência de um ciclo para se obter a instrução da memória, a mesma necessita de mais um ciclo para ser escrita no registrador de pipeline e se propagar para o estágio de decodificação. Sendo essa latência de um ciclo então utilizada para a decodificação da instrução RVC. Há

Figura 3.9 – Diagrama de blocos do pipeline de cinco estágios, IF, ID, EX, MEM e WB. O primeiro estágio, de busca de instrução possuí um ciclo de latência devido a característica síncrona dos blocos de RAM da FPGA Spartan-6, sendo utilizado em sua saída um decodificador para instruções de 16 bits da extensão C, representado pelo bloco RVC DEC.



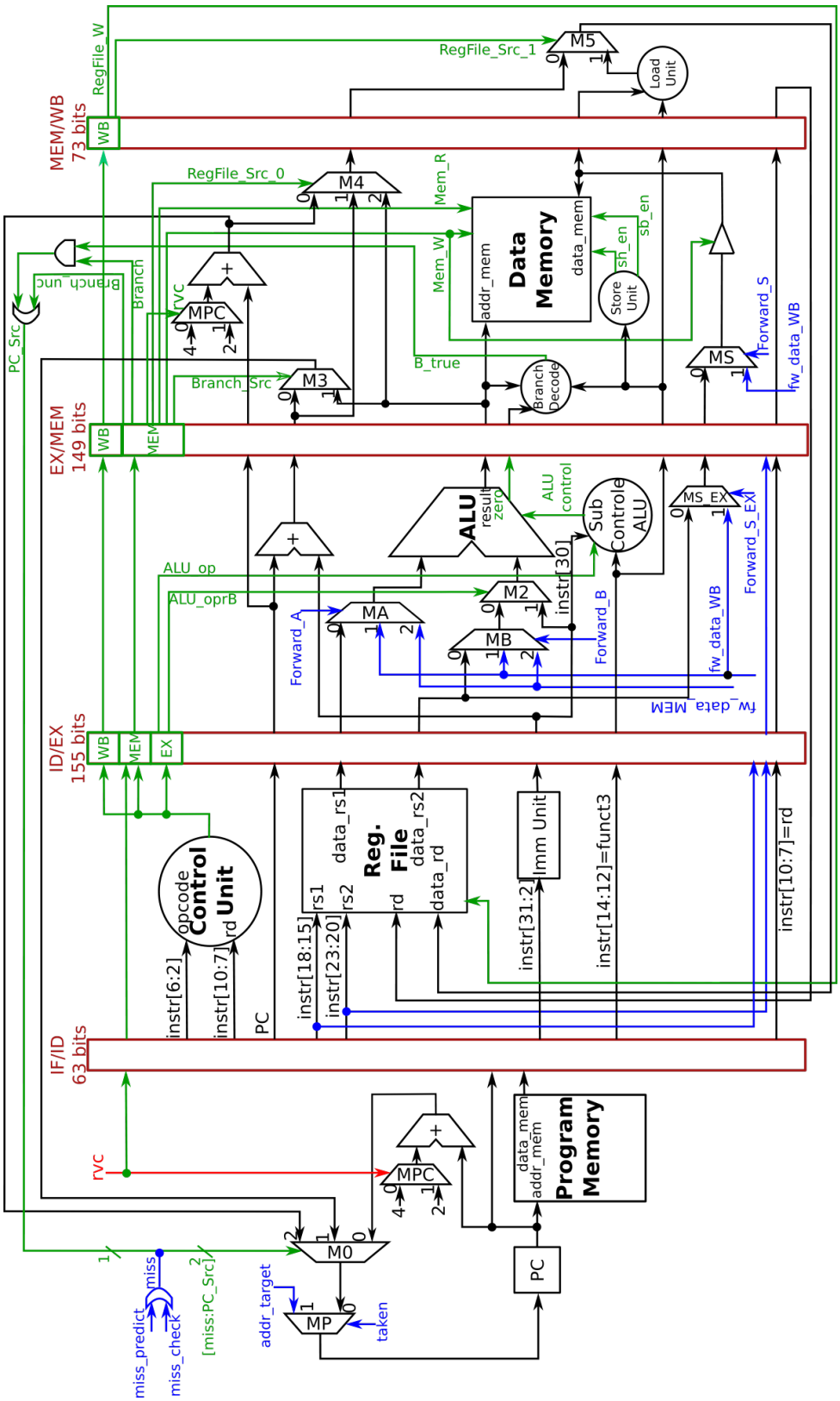
Fonte: Autor.

situações em que se obtém ainda um sétimo ciclo devido a acessos na memória no quarto estágio.

Uma representação detalhada do caminho de dados pode ser observado na Figura 3.10, sendo importante ressaltar que apesar de se apresentar duas memórias, elas utilizam o mesmo barramento de dados e endereço, portanto um acesso no quarto estágio causa stall nos estágios anteriores do pipeline. Ocultou-se as unidades de resoluções para dependências de dados verdadeiros (forward), e também as dependências de controle (preditor de desvios), a fim de deixar mais clara a visualização do caminho de dados. Porém manteve-se os sinais de saída dos blocos ocultados conectados aos componentes destinos, não deixando entradas flutuando no diagrama.

O sinal "rvc" no estágio um é proveniente da unidade de decodificação da instrução de 16 bits, exterior ao núcleo, controlando assim o incremento de PC. Quando em nível lógico 1, indica que a instrução acessada da memória é da extensão C, e o incremento do PC deve ser por 2, e não 4, sendo essa informação propagada até o estágio quatro (MEM), para ser reutilizado. Todas as instruções de desvio, condicional e incondicional, utilizarem o valor atual do PC no cálculo do endereço alvo, e não o próximo (incrementado), por essa razão é passado ao estágio seguinte o valor atual de PC. Porém tem-se um único caso em que o valor do PC incrementado deve ser utilizado, sendo na instrução JALR, em que enquanto se calcula o valor do endereço alvo com o atual do PC somado ao imediato, armazena-se no banco de registradores o PC seguinte ao da instrução de desvio, ou seja, o incrementado. Se realiza novamente o incremento no estágio 4, criando-se assim uma redundância em relação ao seu valor nos estágios do pipeline, e necessitando da ação do

Figura 3.10 – Diagrama do caminho de dados do núcleo pipeline. A unidade de hazards de dado e controle estão ocultadas, demonstrando-se apenas as conexões de suas saídas.



Fonte: Autor.

sinal "rvc" para controlar o incremento em mais de um momento, no estágio IF e MEM.

No segundo estágio do pipeline, três componentes operam em paralelo, sendo eles a unidade de controle, banco de registradores, e a unidade de imediato. A unidade de controle é responsável pela decodificação do opcode da instrução (campo [6:0]), aonde os 2 LSB do mesmo são ignorados, já que após a verificação do tamanho da instrução na etapa de busca (estágio IF) não servem de mais nada. Também se utiliza o endereço de "rd" como entrada, destino de escrita do banco de registradores, quando o mesmo for igual a 0, o sinal "RegFile_W" é desabilitado, nível lógico 0, pois a escrita deve ser ignorada. A fim de ignorar apenas a escrita no registrador x0, não seria necessária essa ação, pois o registrador x0 não seria implementado, utilizando-se uma conexão direta com o GND, por exemplo. Porém como descrito na Seção 3.2.2.5, essa ação é fundamental para prevenir comportamentos errados por parte da unidade de forward. O campo funct3 da instrução (bits [14:12]) é propagado ao longo do pipeline, para ser utilizado como auxílio na decodificação de instruções por meio de subunidades de controle, que se totalizam em quatro, subunidade de controle da ALU, decodificador de branch (branch decode), unidade de store (store unit) e unidade de load (load unit).

Os diversos multiplexadores ao longo do núcleo possuem como função:

• MUX PC: define o valor de incremento do PC entre 4 (para instrução da base E) e 2 (para instrução da extensão C), controlado por "rvc", sendo que o mesmo está presente tanto no estágio IF quanto MEM, devido a redundância do uso de PC em instruções de desvio;

• MUX P: define o endereço a ser escrito no PC, entre o da saída do MUX 0, ou o endereço alvo do preditor, sendo controlado pelo sinal "taken";

• MUX 0: define o endereço a ser escrito no PC, entre o seu incremento, endereço alvo calculado no estágio MEM, ou seu endereço incrementado para ajuste de seu valor, no caso do preditor agir como taken, e se provar errado, o sinal de controle do multiplexador é de 2 bits, o MSB do preditor, e o LSB sendo o sinal "PC_Src";

• MUX A: define o operando A da ALU, entre o valor lido do banco de registradores, dado realimentado do estágio MEM, ou então realimentado do estágio WB, controlado pelo sinal "Forward_A";

• MUX B: define o operando B da ALU, entre o valor lido do banco de registradores, dado realimentado do estágio MEM, ou então realimentado do estágio WB, controlado pelo sinal "Forward_B", possuí menor prioridade em relação ao sinal "ALU_oprB" do MUX 2;

• MUX 2: define o operando B da ALU entre a saída do MUX B, e o imediato da instrução (já decodificado e estendido para 32 bits), controlado pelo sinal "ALU_oprB";

• MUX S EX: define o dado a ser escrito na memória em instruções de store, entre o valor lido do endereço "rs2" no banco de registradores, ou o realimentado do estágio WB, controlado pelo sinal "Forward_S_EX";

• MUX 3: define o endereço alvo de desvio entre o de PC somado ao imediato (que possuí um somador dedicado), ou então do registrador lido do banco somado ao imediato (somado na ALU), controlado por "Branch_Src";

• MUX 4: define um dos possíveis dados a serem escritos no banco de registradores entre o valor de PC incrementado, o valor de PC somado ao imediato (utilizado em instruções AUIPC), e a saída da ALU, controlado por "RegFile_Src_0";

• MUX S: define o dado a ser escrito na memória em instruções de store, entre o valor lido do endereço "rs2" no banco de registradores, ou o realimentado do estágio WB, controlado pelo sinal "Forward_S";

• MUX 5: define o dado a ser escrito no banco de registradores entre o valor de saída do MUX 4, e o dado lido da memória no estágio anterior (após passar pela unidade de decodificação load unite), controlado por "RegFile_Src_1".

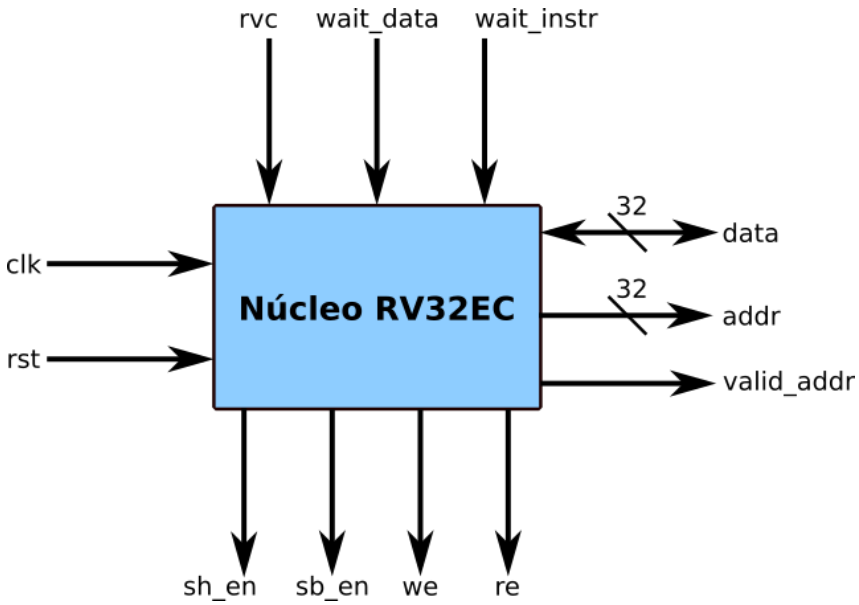
Na Figura 3.11 apresenta-se uma visão de topo do núcleo, com as suas entradas e saídas de controle, dados, e endereço. O núcleo responde a borda de subida do sinal de relógio, e o seu reset é síncrono, em nível 1. A Tabela 3.7 possui uma breve descrição das portas do núcleo. O sinal valid_addr ativo em nível 0, indica que o endereço é válido, a geração do sinal está conectado diretamente a ocorrência de flush do pipeline devido a desvios condicionais e incondicionais. Todos os sinais relacionados a escrita (sh_en, sb_en, we) e a leitura (re) da memória são ativos em nível 1, sendo os sinais "we"e "re"correspondentes aos Mem_W e Mem_R, respectivamente, do caminho de dados. Quando se tem as portas de entrada de espera pela memória (wait_data e wait_instr) em nível 1, se trava o pipeline de acordo com a lógica de cada um. O sinal "rvc"é fornecido pela unidade de decodificação das instruções RVC, localizada no decodificador de controle para acessos a memória, externo ao núcleo.

Tabela 3.7 – Portas de entrada e saída do núcleo.

Porta	Descrição
clk	sinal de relógio
rst	sinal de reset
rvc	indica se a instrução é de 16 bits ou não
wait_data	sinal de espera pelo dado da memória
wait_instr	sinal de espera pela instrução lida da memória
sb_en e sh_en	habilitação de escrita por byte ou half word da memória
we e re	habilitação de escrita e leitura da memória
valid_addr	indica quando o endereço no barramento é válido

Fonte: Autor.

Figura 3.11 – Visão de topo do núcleo pipeline RV32EC, com um barramento de dados bidirecional, um barramento de saída de endereço, e seus sinais de controle de entrada e saída.



Fonte: Autor.

3.2.2.1 Banco de Registradores

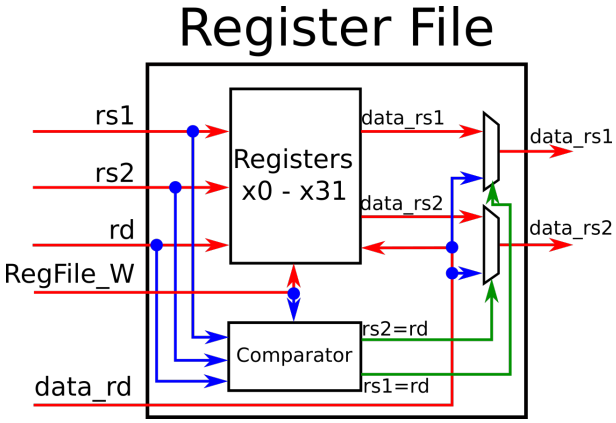
Como destaque da base E do RISC-V, o banco de registradores possui apenas 16 de seus 32 registradores de uso geral disponíveis, referenciados por x0 até x15. Possuí duas portas de entrada para endereços a serem acessados do banco, ambas de 4 bits, "rs1" e "rs2", assim como uma porta com o endereço alvo de escrita, "rd". Suas saídas "data_rs1" e "data_rs2" com 32 bits de tamanho, correspondem aos dados lidos de forma assíncrona, nos endereços das portas de leitura, e a porta de dado de entrada também de 32 bits, "data_rd", corresponde a informação que se deseja armazenar no endereço "rd", este de forma síncrona na borda de subida do relógio.

O registrador x0, por definição da ABI do RISC-V é a constante 0, sendo assim qualquer escrita no mesmo é ignorada. O banco sendo localizado no segundo estágio do pipeline (ID), realiza a leitura durante esta mesma etapa, porém a escrita de dados ocorre apenas no quinto estágio, WB, após a propagação do sinal de controle "RegFile_W". Por essa razão, caso um dado esteja sendo escrito no banco, e seu endereço "rd" coincida com um dos de leitura, é necessário a lógica de forward, a fim de garantir a propagação do dado correto. O banco de registradores realiza essa lógica internamente ao comparar os endereços das portas de leitura, com o destino de escrita, caso se confirme a igualdade, o dado de escrita é multiplexado diretamente a porta de leitura correspondente, isso apenas se o sinal "RegFile_W" estiver em nível 1 (ativo).

Diversas pseudo-instruções do RISC-V utilizam do registrador x0 na sua execução,

logo é necessário que não somente o banco de registradores ignore a escrita no endereço 0, como também que o sinal "RegFile_W" esteja em nível 0 quando o mesmo é alvo. Essa necessidade surge da possibilidade do forward repassar para uma das portas de saída o dado de escrita, que deve ser ignorado no registrador x0. Porém a escrita ("RegFile_W = 1") está habilitada, ocasionando na propagação de um dado aleatório aos estágios seguintes, que deveria ser a constante 0. A Figura 3.12 apresenta um diagrama interno do banco de registradores, com sua lógica de forward.

Figura 3.12 – Lógica de forward interno do banco de registradores.



Fonte: Autor.

3.2.2.2 Unidade Imediato

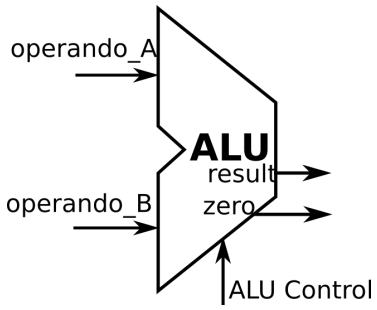
A unidade imediato, localizada no segundo estágio do pipeline, ID, tem por finalidade decodificar o imediato a ser utilizado nas instruções, pois seus bits estão embaralhados ao longo dos diferentes formatos da base E do RISC-V. Internamente, utiliza-se de multiplexadores, que tem por finalidade gerar o sinal desejado, sendo que o imediato das instruções da base E sempre serão estendidos para 32 bits por meio do MSB. O campo da instruções com os bits [31:7] representa todos os bits possíveis de se ter em um sinal imediato, sendo que o opcode da instrução (bits [6:2]) são utilizados para decodifica-lo, logo seu funcionamento independe da unidade de controle principal, podendo operar em paralelo no estágio ID. O decodificador é organizado conforme o tipo de imediato do manual a nível de usuário do RISC-V (WATERMAN; ASANOVIć, 2017a, p. 12).

3.2.2.3 Unidade Lógica e Aritmética

Componente responsáveis pela execução de operações lógicas e aritméticas, sobre dois operando, A e B de 32 bits, com uma saída resultante também de 32 bits, com um sinal de controle "zero" indicando quando a operação resultou em 0, e sem verificação de overflow. Em instruções que utilizam de sinais imediato, o mesmo sempre é o operando B, incluindo em operações de deslocamento, em que o imediato contém a informação relativa ao tamanho do deslocamento. A Figura 3.13 apresenta as portas de entrada e saída da ALU. Suas operações são todas realizadas em um único ciclo de relógio, sendo visualizado na Tabela 3.8 a codificação de suas possíveis operações, de acordo com o sinal de controle ALU Control. Quando seu valor corresponde a 1010, o operando B é multiplexado para a saída, pois o mesmo corresponde ao imediato da instrução LUI (load upper immediate) que deve ser armazenado no banco de registradores.

Internamente a unidade contém três comparadores, um de igualdade do resultado a 0, e outros dois que verificam se o operando B é maior ou igual ao operando A, com e sem sinal. Os comparadores de magnitude dos operando são utilizados na execução das instruções STL[U] e STLI[U] (set less than e set less than immediate), que quando verdadeiro, resulta em 1 na saída de operações da ALU. O seu resultado é reaproveitado no estágio seguinte do pipeline, MEM, para a resolução de instruções de desvio condicional BGE[U] e BLT[U] (branch if greater or equal e branch if less than). A saída zero da ALU, que é consequência do comparador a 0 da operação executada, é utilizada no estágio MEM para a resolução das demais instruções de branch, BEQ e BNE (branch if equal e branch if not equal).

Figura 3.13 – Bloco lógico da ALU.



Fonte: Autor.

3.2.2.4 Dependências Estruturais

Devido o uso de um barramento único para acessos a memória, surge a necessidade de se travar o pipeline no estágio IF quando se está lendo ou escrevendo um dado no

Tabela 3.8 – Codificação das operações da ALU, valores maiores que 1010 resultam em adição.

ALU Control	Operação
0000	ADD
0001	SUB
0010	SLL
0011	SRL
0100	XOR
0101	SRA
0110	OR
0111	AND
1000	SLT
1001	SLTU
1010	opr_B(LUI)
>1010	ADD

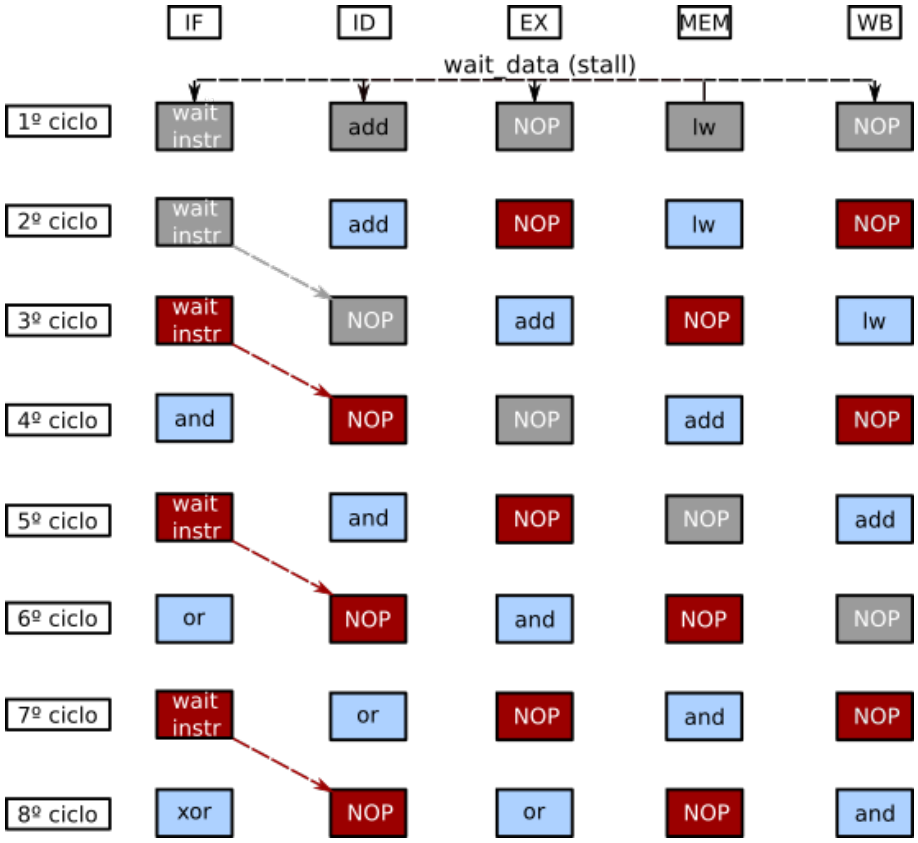
Fonte: Autor.

estágio MEM. Essa dependência resulta em uma perda de um ciclo de relógio, sendo que com a latência de um ciclo nos acessos aos blocos de RAM da FPGA Spartan-6, obtém-se ainda mais um ciclo de perda. A Figura 3.14 demonstra o pipeline em execução, com uma memória síncrona de um ciclo de acesso, percebendo-se que durante a espera pela memória no estágio IF, é passado para o estágio ID uma instrução NOP, pois não se tem a necessidade de parar a execução de todo o pipeline. Já para acessos no estágio MEM, todo o pipeline é travado, impedindo qualquer escrita nos seus registradores, incluindo no estágio WB, que acaba por repetir a última operação de escrita no banco de registradores (caso habilitada), não alterando em nada o núcleo.

No programa executado, ao travar o estágio IF por dois ciclos quando se tem um acesso no estágio MEM, ainda se tem mais um ciclo de espera antes da busca da próxima instrução. Como resultado dessa execução demonstrada, o número de instruções por ciclo obtido é de apenas 0,375, reduzindo significativamente o desempenho do núcleo. Observando-se que, devido a latência de um ciclo de busca por instruções, se tem a presença de pelo menos um NOP inserido por espera a memória entre instruções a serem executadas. Assim, a instrução no estágio WB no primeiro ciclo da execução exemplificada corresponde a um NOP alocado devido a espera pela busca justamente da instrução LW quando a mesma se encontrava no estágio IF.

O núcleo possuí dois sinais de entrada para controlar seus registradores em relação a latência de acesso, "wait_data" e "wait_instr". O primeiro trava a escrita sobre todos os registradores, e está associado a qualquer busca de dados, ou seja, não somente a informações armazenadas na memória de dados, mas também de constantes que podem estar contidas na memória de programa. Já o sinal "wait_instr", como o nome sugere é utilizado unicamente para inserir NOP no estágio seguinte enquanto espera pela instrução

Figura 3.14 – Execução de um programa com perdas de ciclo devido a espera por dados da memória.



Fonte: Autor.

da memória de programa, sendo que ela trava não somente o registrador entre os estágios IF e ID, e sim qualquer elemento de memória presente no primeiro estágio. Que se resume a lógica interna do preditor de desvios dinâmicos e o contador de programa (PC). Ambos os sinais de espera pela memória são ativos em nível lógico 1. A instrução NOP é transferida de acordo com sua codificação completa, visto que no estágio 1 não se tem ainda nenhum sinal de controle para desabilitar, o que simplificaria o processo.

3.2.2.5 Dependências de Dados

A unidade de forward, para resolução de dependências de dados verdadeiras, está representada dentro do caminho de dados conforma a Figura 3.15, ocultando-se diversos outros componentes para uma melhor visualização. Outros tipos de dependências de dados não ocorrem, consequência da execução em ordem com emissão de uma instrução por ciclo do núcleo. Seu funcionamento realimenta o resultado de instruções, antes das mesmas serem escritas no banco de registradores no quinto estágio, WB. As possibilidades são do estágio WB para o MEM ou EX, e do estágio MEM para o EX. Nota-se que

o feedback do estágio WB para o MEM está presente unicamente para a execução de instruções de store, representando o dado a ser escrito pela saída do multiplexador MS. Sendo a mesma lógica necessária no estágio EX para a escrita na memória, com o dado selecionado pelo MUX MS EX.

A lógica da unidade de forward se baseia em comparadores, caso o registrador destino de um estágio mais avançado do pipeline coincida com um dos registradores fonte de leitura do atual, e a escrita (RegFile_W) esteja habilitada no estágio avançado, um sinal multiplexa o dado de realimentação para ser utilizado, no lugar do desatualizado que estava circulando pelo estágio atual. No estágio ID ocorre a comparação do registrador destino de escrita com 0, logo se o alvo for x0, que corresponde a constante 0, o sinal de habilitação de escrita é desabilitado. Assim não tem-se a necessidade da unidade de forward (e também do forward interno do banco de registradores) verificar se o registrador destino é igual a 0, pois a escrita estará desabilitada, e o forward será ignorado.

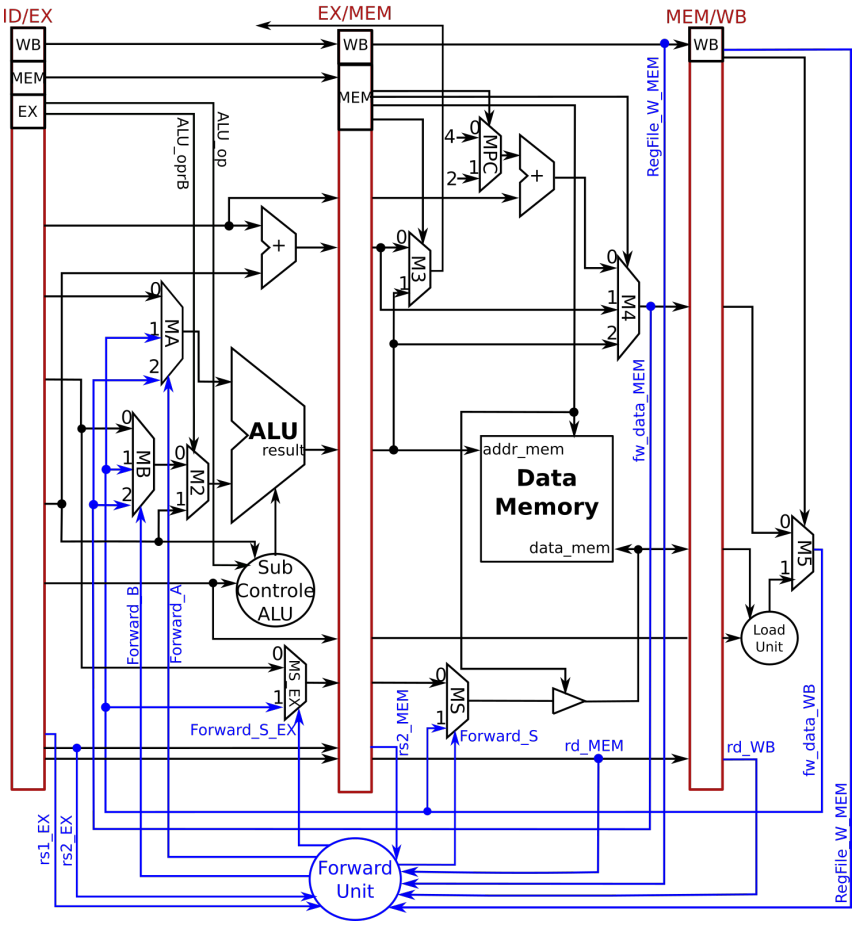
Com o uso da técnica de forwarding, perdas de ciclos de relógio por conta de dependências de dados verdadeiras são resolvidas, com exceção do caso de instruções de load seguidas por uma que utilize o dado a ser buscado da memória. Sua detecção é realizada no estágio de decodificação, inserindo então uma instrução NOP no estágio EX quando necessário, e travando a atualização do estágio anterior, IF, conforme demonstra a Figura 3.16. Outras unidades foram ocultadas, mostrando apenas os sinais necessários de outros estágios, habilitação de leitura no estágio MEM (MEM_R_EX) e registrador destino do estágio EX (rd_EX).

A instrução NOP é inserida no estágio seguinte (EX) por meio de uma operação AND com os sinais de controle, sendo o mesmo sinal utilizado nesta operação, responsável por um stall no estágio IF. Para instruções que fazem leitura do registrador destino do último estágio, WB, se utiliza da lógica de forward internamente no banco de registradores, o mesmo compara os registradores alvos de leitura e de escrita, no caso de serem iguais multiplexa o dado na porta de escrita para a saída correspondente a da leitura, conforme descrito na Seção 3.2.2.1.

3.2.2.6 Dependências de Controle

Para a predição de desvios condicionais, se optou pelo uso do preditor dinâmico bimodal, que conforme demonstrado por Arora, Kotecha e Samyal (2013), apresenta uma melhor taxa de acertos que os demais métodos como o preditor dinâmico de 1 bit, ou preditor estático. Sua complexidade também foi outro fator decisivo, pois o mesmo opera sobre uma máquina de estados de 2 bits, sendo assim uma implementação relativamente simples. A fim de manter essa simplicidade na implementação, a branch history table foi organizada com um mapeamento do tipo direto, sendo então descartado o uso de algo-

Figura 3.15 – Unidade de forward do núcleo pipeline. Funciona comparando o registrador destino dos estágios seguintes, com os registradores fonte dos estágios anteriores. Caso um dado mais novo que o lido do banco de registradores esteja disponível, os sinais Forward_A e Forward_B selecionam o correspondente. Dando prioridade para o estágio MEM (dado mais recente). Na execução de instrução store, o forward é controlado pelos sinais Forward_S_EX e Forward_S.

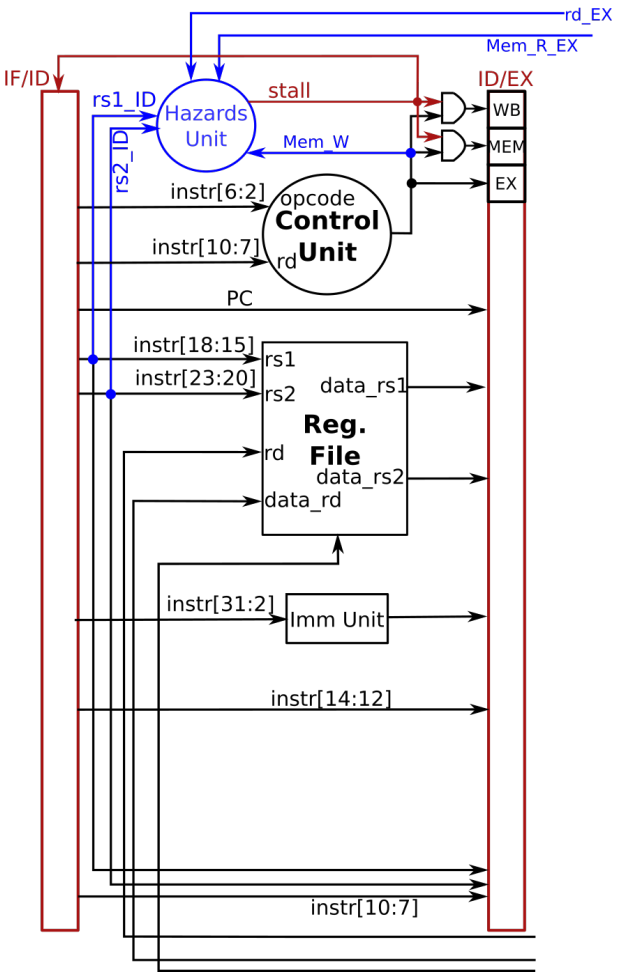


Fonte: Autor.

ritmos de substituição por hardware, a custo de desempenho. O tamanho da tabela foi determinado de acordo com os blocos de RAM disponíveis na FPGA Spartan-6 em uso, com um total de 32 blocos de 18 kbits, procurou-se utilizar o menor número possível de BRAMs, porém mantendo um número de linhas de acesso na tabela de 512 a 2048.

Das possíveis configurações de largura e profundidade de um bloco de RAM, disponível para consulta no manual Xilinx (2011b), ao utilizar-se um bloco com largura de 32 bits, para armazenamento do endereço alvo de desvio, se obtém um bloco de 32x512, aproveitando 16 kbits do total de 18kbits do bloco. Com um bloco ocupado unicamente para os endereços alvos, se utilizou outro para armazenamento da tag, dois bits de previsão de estado, e um bit de validade que inicializa em estado lógico baixo no reset do sistema, formando um bloco de 25x512. O tamanho da tag se resumiu a 22 bits, ao remover os 9 bits inferiores para indexação da tabela, e mais 1 bit devido ao alinhamento das instruções,

Figura 3.16 – Unidade de detecção de dependências de dados verdadeiras, causadas por instruções de load e uma outra em sequência que utilize de seu resultado.



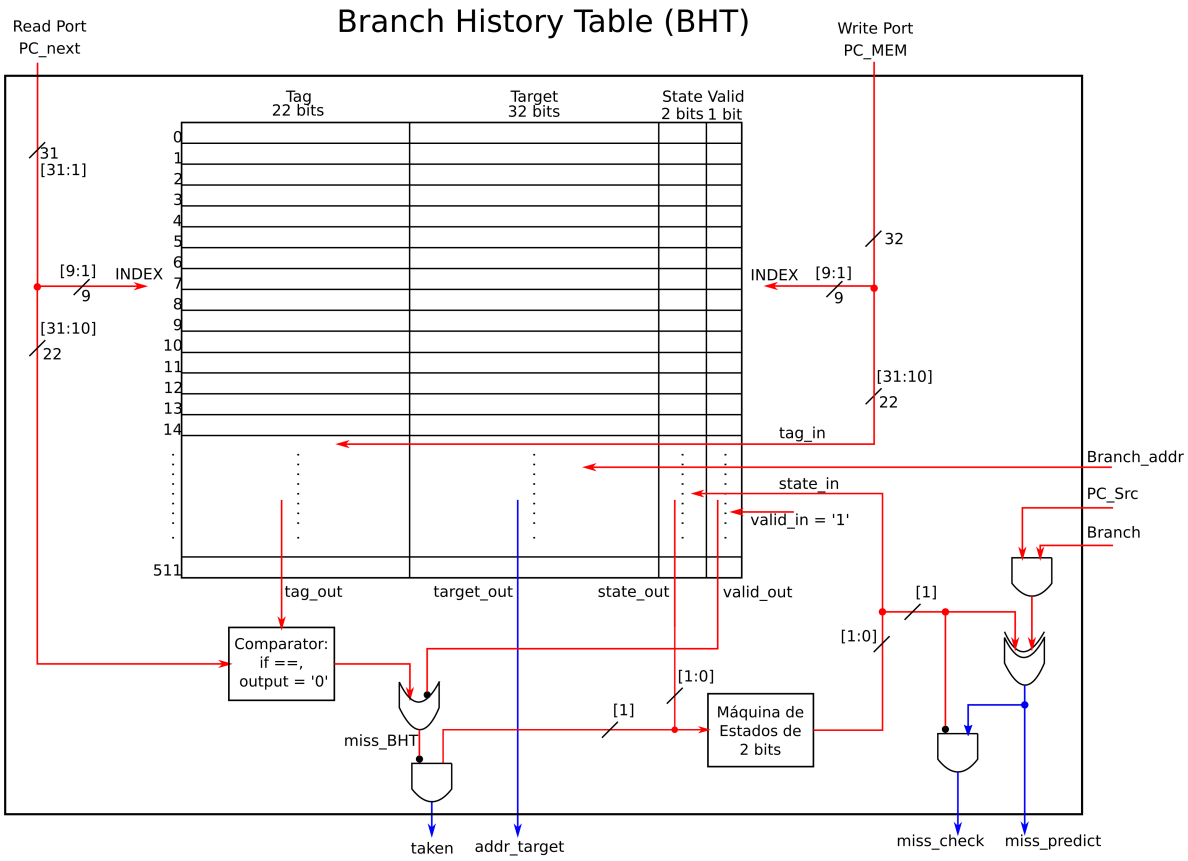
Fonte: Autor.

ou seja, o bit 0 de PC possuí sempre nível 0. Com a BHT de 512 linhas apenas, optou-se por não armazenar desvios incondicionais (jumps) na mesma, devido a baixa ocorrência das mesmas em comparações a branch, muito utilizados em loops. Logo, a ocorrência de desvios incondicionais sempre gera perdas, causando flush no pipeline do estágio 1 ao 4.

A Figura 3.17 apresenta o esquemático da tabela projetada para o preditor, as saídas são os sinais "miss_predict"e "miss_check", que são complementares. Para garantir que a instrução é do tipo de desvio condicional, e evitar o armazenamento de instruções do tipo jump, se utiliza o sinal "Branch", proveniente também do quarto estágio, para verificar o tipo de instrução de desvio, e "PC_Src" que contém o resultado do teste do branch.

O diagrama do caminho de dados apresentando a unidade de predição está contido na Figura 3.18, ignorando-se os estágios seguintes ao primeiro (IF), mostrando apenas os sinais necessários. O endereço PC no estágio MEM é utilizado como indexação da tabela do preditor, para escrita do endereço alvo e/ou atualização do estado. O seu valor com

Figura 3.17 – Tabela de histórico de desvios, possuí uma porta de leitura indexada pelo endereço de entrada do PC no estágio IF, e uma de escrita pelo endereço de PC no estágio MEM. Os campos tag, state e valid, correspondem a um bloco de RAM, e o campo target a um segundo bloco.



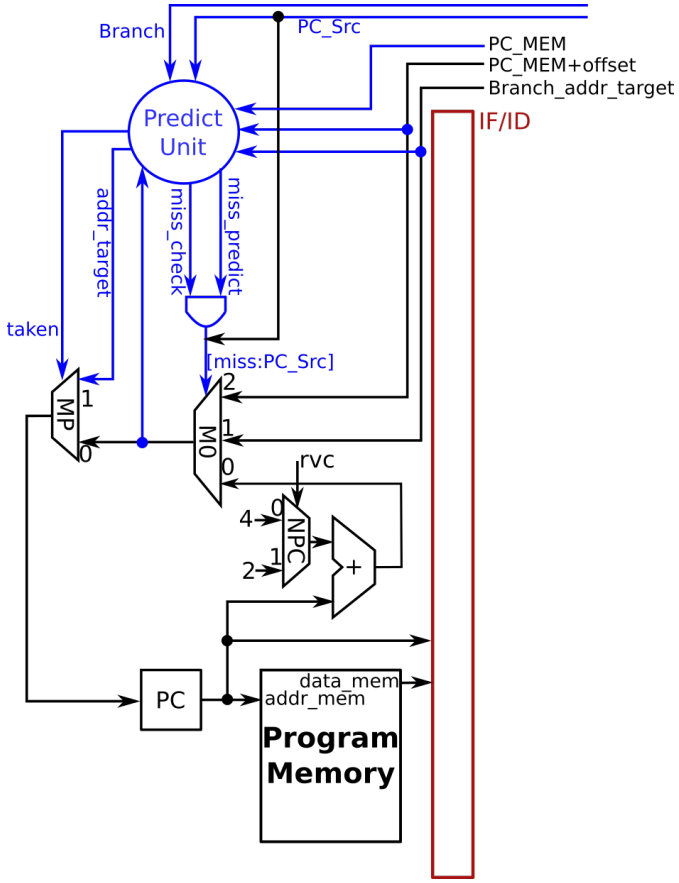
Fonte: Autor.

incremento do quarto estágio também é utilizado para correção do PC em ações erradas da unidade. O sinal que controla o multiplexador M0 tem como prioridade a ação do preditor, logo o bit mais significativo é o resultado da operação AND entre os sinais "miss_check" e "miss_predict".

Devido a característica síncrona dos blocos de RAM, o endereço a ser utilizado para procura na tabela não é o da saída do registrador de PC, e sim o de sua entrada, pois com a leitura síncrona se obtém uma latência de um ciclo na saída de dados. Utilizando-se a entrada de PC como indexação da tabela, se mantém a funcionalidade do preditor sem a necessidade de inserir instruções NOP no registrador de pipeline do estágio 1 (Instruction Fetch) enquanto se aguarda a saída da tabela de branch. Essa decisão considerou tanto o uso de memórias externas ao núcleo (ROM e RAM) assíncronas, que foram utilizadas nas etapas iniciais do projeto para simulações e testes do núcleo devido a simplicidade, quanto para síncronas, implementadas em etapas finais como blocos de RAM. Como resultado, o controle do preditor deve ser robusto o suficiente para manter o valor da saída

do multiplexador M0 estável, garantindo o funcionamento do preditor.

Figura 3.18 – Unidade de predição no estágio um do pipeline.



Fonte: Autor.

Os sinais de controle de saída do preditor se comportam conforme:

• taken: quando em nível 0 indica que a predição da linha acessada na BHT é para não desviar ao endereço "addr_target", e para nível 1, para desviar;

• miss_check: quando em nível 1 indica que um desvio que ocorreu anteriormente tinha sido previsto como taken, e o PC deve ser ajustado para o valor de "PC_MEM+offset", para 0, se corrige o valor de PC para o calculado "Branch_addr_target", esse sinal é ignorado quando "miss_predict" está em nível 0;

• miss_predict: quando em nível 1 indica que o preditor errou uma ação anterior, e deve ser corrigido de acordo com o indicado pelo sinal complementar "miss_check", em nível 0 procede normalmente com o controle sendo do sinal "PC_Src" sobre o MUX 0.

3.2.2.7 Unidade de Controle

A unidade principal de controle do núcleo esta localizada no segundo estágio, ID, possuindo duas entradas, o campo opcode da instrução que corresponde aos bits [6:0], e o campo rd, registrador destino de escrita que está associado aos bits [10:7]. Como os 2 LSB do opcode contribuem apenas para a decodificação do tamanho da instrução, após seu uso na determinação do seu tamanho entre 32 bits (base E) ou 16 bits (extensão C), no estágio de busca, IF, os mesmos podem ser descartados, e nem se quer são salvos no registrador de pipeline. De forma semelhante, com o uso da base E não se tem a necessidade de utilizar o MSB dos campos de acesso aos registradores do banco, assim ignora-se o bit [11] do endereço rd.

O uso do registrador rd na unidade principal de controle é devido ao fato de ignorar-se a escrita no registrador x0 do banco, assim, quando o mesmo for igual a 0, o sinal de controle responsável pela escrita, "RegFile_W", é desabilitado ao coloca-lo em nível 0. Essa lógica é necessária para evitar conflitos no uso da unidade de forward, conforme descrito na Seção 3.2.2.5.

Seu hardware é composto apenas por lógica combinacional, sendo essencialmente um decodificador, e de simples implementação. Os sinais de controle que são gerados de sua ação, totalizam em 10, com as seguintes características:

• ALU_oprB: controla o MUX 2, escolhendo o operando B da ALU entre a saída do MUX B, para o nível 0, e o imediato da instrução, quando em nível 1;

• ALU_op: possuindo 2 bits, é utilizado para definir a operação que será executada na ALU, sua codificação na subunidade de controle da ALU é descrita na Seção 3.2.2.8;

• Branch_Src: escolhe o endereço alvo de instruções de desvio, para nível 0 utiliza o valor de PC somado ao imediato, e quando em nível 1 o valor de um registrador do banco somado ao imediato (instrução JALR unicamente);

• Branch: quando em nível 1, indica que a instrução em execução é um desvio condicional (branch);

• Branch_un: quando em nível 1, indica que a instrução em execução é um desvio incondicional (jump);

• Mem_W: quando em nível 1, habilita o buffer tri-state alocado na saída do MUX S, sendo o sinal enviado também para a memória, seu valor nunca está ativo simultaneamente com Mem_R;

• Mem_R: quando em nível 1, habilita a leitura da memória, enviando o sinal para a memória, seu valor nunca está ativo simultaneamente com Mem_W;

• RegFile_Src_0: composto por 2 bits, escolhe um dos possíveis dados a ser armazenado no banco de registradores, sendo o valor de PC somado ao incremento quando em nível 00, o valor de PC somado ao imediato para nível 01, ou então o resultado obtido

na execução da ALU, quando em nível 10;

• RegFile_Src_1: conceitualmente é o terceiro e mais significativo bit de RegFile_Src_0, escolhendo definitivamente o dado de escrita no banco de registradores, entre a saída do MUX 4 para nível lógico 0, ou então um dado acessado da memória em nível lógico 1;

• RegFile_W: habilitação de escrita no banco de registradores quando em nível lógico 1.

O sinal RegFile_Src_1 de fato, é o sinal Mem_R renomeado e propagado ainda mais um estágio dentro do pipeline, pois a escolha de se armazenar um dado lido da memória no banco só ocorre quando o mesmo está em nível 1, reaproveitando-se assim o sinal. Alguns sinais de controle são gerados nos estágios seguintes, pelas subunidades da ALU, branch, store e load, abordadas na Seção 3.2.2.8. Outros possuem origens diferentes, sendo eles:

• PC_Src: quando em nível 1, seleciona o endereço alvo de desvio no MUX 0, sendo sua prioridade baixa em relação aos sinais de "miss" do preditor, sua lógica é uma operação OR entre os sinais Branch_unc e o resultado do teste de branch;

• rvc: sendo um sinal de entrada no núcleo, indica quando uma instrução acessada no estágio IF é de 16 bits, diferenciando o incremento do PC entre 2 ou 4, para nível lógico 1 e 0, respectivamente.

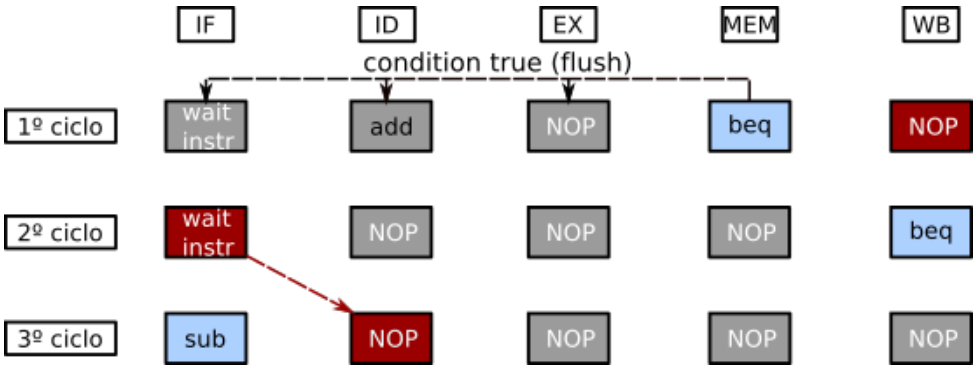
O sinal valid_addr de saída do núcleo não é gerado pela unidade de controle principal, mas é um sinal de grande importância em sua execução, sendo seu valor atribuído de acordo com os desvios condicionais e incondicionais de instruções executadas. A Figura 3.19 exemplifica seu uso, para um código simples, como:

```
start:
beq  x1,x2,target
add  x1,x2,x3
add  x4,x5,x6
target:
sub  x1,x2,x3
sub  x4,x5,x6
```

aonde no momento em que se está decidido que a instrução de branch irá desviar o valor de PC (supondo que a condição teste de positivo), no estágio IF já está sendo buscada uma instrução da memória. Como o controle da memória já está contabilizando o ciclo de latência de acesso, por mais que se de um flush no pipeline e descarte a execução atual dos estágios, o núcleo irá receber a instrução que não deveria ter buscado, e armazena-lá no registrador entre IF e ID. Para isso, o sinal de validade do endereço indicá a memória

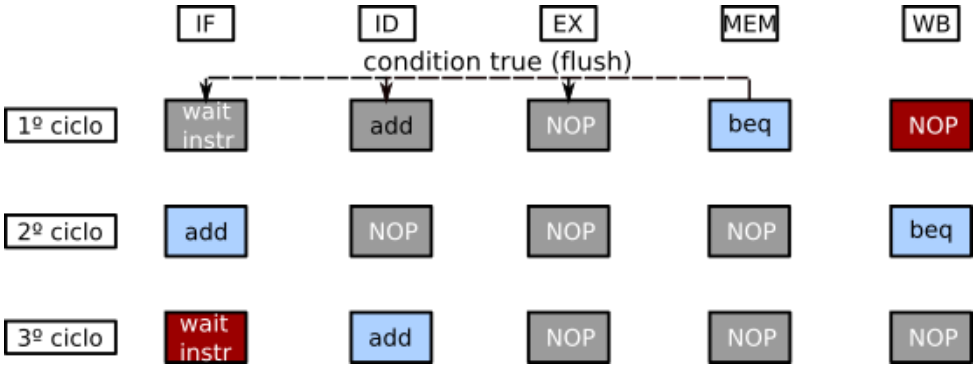
para não considerar aquele acesso, desabilitando o chip de memória correspondente. Supondo a ausência do sinal de validade, iria ocorrer a execução da instrução ADD após o BEQ, como demonstra a Figura 3.20.

Figura 3.19 – Execução de um programa exemplificando o uso do sinal valid_addr.



Fonte: Autor.

Figura 3.20 – Execução de um programa exemplificando o impacto da ausência do sinal valid_addr.



Fonte: Autor.

3.2.2.8 Subunidades de Controle

As unidades de store a load, presentes no estágio MEM e WB, respectivamente, funcionam de forma semelhante, utilizam do campo funct3 da instrução para definir se o acesso a memória é de uma palavra inteira (32 bits), meia palavra (16 bits) ou então um byte apenas, sendo que para load ainda se obtém diferenciação entre dados com sinal e sem sinal. A unidade de store simplesmente envia dois sinais de controle para a memória, "sh_en" e "sb_en", como os nomes indicam, habilitação de meia palavra e habilitação de byte. Somente um pode ser ativo por vez em operações de escrita, ativos em nível 1, indicando a memória para armazenar somente os LSB, e ignorando a escrita dos demais

campos do dado enviado. No caso de uma escrita de palavra inteira, ambos os sinais estão em nível 0.

A unidade de load recebendo o dado de 32 bits da memória, irá manter o mesmo inalterado se for uma instrução de acesso a palavra. Porém caso se tenha uma instrução LB/LH (load byte ou load half), o MSB do dado é estendido até uma palavra inteira, ignorando-se qualquer valor que ali tivesse, o mesmo se aplica para LBU/LHU (unsigned), porém estendendo com 0. A Tabela 3.9 possui a codificação do campo funct3 da instrução, em relação as unidades de store e load. Para instruções de store, apenas os bits [13:12] de funct3 são utilizados, já para load o campo inteiro é necessário.

Tabela 3.9 – Codificação do campo funct3 de instruções do tipo store e load.

funct3[13:12]	Store
00	store byte (sb_en = 1, sh_en = 0, Mem_W = 1)
01	store half (sb_en = 0, sh_en = 1, Mem_W = 1)
1x	store word (sb_en = 0, sh_en = 0, Mem_W = 1)
funct3[14:12]	**Load**
000	load byte signed
001	load half signed
x1x	load word signed
100	load byte unsigned
101	load half unsigned

Fonte: Autor.

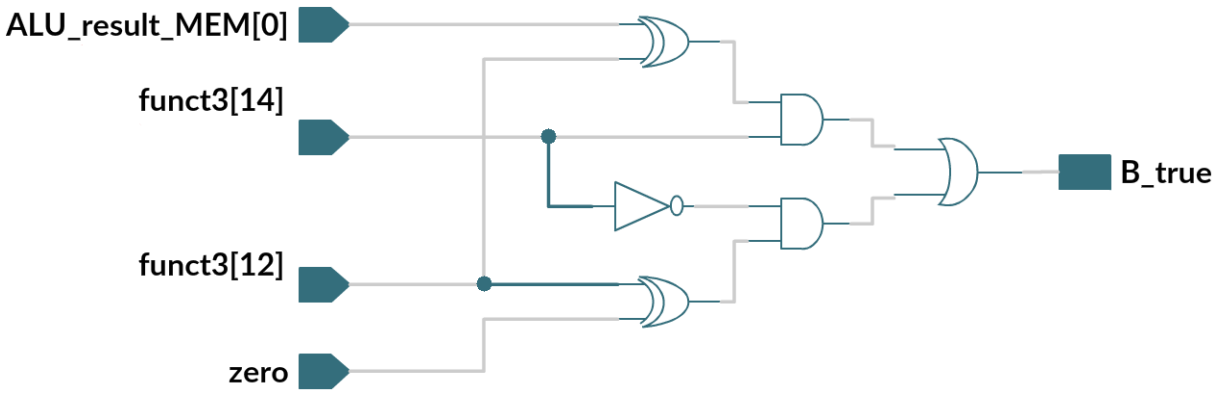
O decodificador de branch é utilizado para identificar o tipo de branch em execução, sendo eles branch if equal (BEQ), branch if not equal (BNE), branch if greater or equal (BGE), ou então branch if less than (BLT), com os últimos dois possuindo variantes para valores sem sinal. O controle é realizado pelo sinal zero (provido pela ALU) e o campo funct3 da instrução, que se propaga através do pipeline, o sinal resultante da decodificação é nomeado de "B_true", considerado ativo em nível 1. No diagrama da Figura 3.21 se visualiza a lógica interna do decodificador, aonde não é utilizado no bit [13] do campo funct3.

Para instruções BEQ e BNE a abordagem é relacionada diretamente ao sinal zero de saída da ALU, quando o mesmo está em nível 1, indica que o resultado de sua operação é igual a 0, e no caso de BEQ, deve desviar, e para BNE, considera-se o valor negado de zero internamente na unidade de branch para essa tomada de decisão. Para que o sinal zero possa ser utilizado nessa diferenciação de BEQ e BNE, a operação realizada pela ALU nessas instruções corresponde a de subtração.

Para instruções BGE e BLT com sinal, se indica para a ALU o uso da operação SLT, na qual ela compara internamente (com sinal) os valores dos operando A e B, e envia para a saída o resultado 1 caso o valor em sua porta A for menor que o de B, do contrário 0. Logo, no estágio MEM (onde se encontra a unidade de branch), o LSB do resultado da

ALU é utilizado da mesma forma que o zero para confirmar se o teste é verdadeiro, e o campo funct3 para diferenciar entre as duas, negando o sinal quando necessário (mesma lógica de BEQ e BNE). O mesmo se aplica para BGEU e BLT (unsigned), porém a ALU utiliza outro sinal de controle em que se considera a comparação sem sinal.

Figura 3.21 – Diagrama de portas lógicas do decodificador de instruções branch.



Fonte: Autor.

A subunidade de controle da ALU possuí três entradas que definem seu valor, indicando a operação da ALU, o sinal de controle "ALU_op" da unidade principal no estágio ID, como as demais subunidades o campo funct3 da instrução, e o bit 30 da instrução, rotulado por funct. O bit funct tem por finalidade diferenciar entre instruções de deslocamento aritmético e deslocamento lógico, no caso, entre SRA e SRL ou SRAI e SRLI, assim como também entre ADD e SUB que possuem o mesmo opcode e funct3. A conexão do bit funct da instrução é proveniente do sinal imediato utilizado no estágio EX, pois ele sempre estará na mesma posição para estas instruções em específico.

O sinal proveniente da unidade principal de controle tem prioridade, possuindo largura de 2 bits, e indicando as ações da Tabela 3.10. Sendo a adição uma operação utilizada na grande maioria das instruções, um dos possíveis valores é atribuído unicamente para a mesma, outro valor é utilizado para a instrução LUI (load upper immediate), em que se faz o resultado da saída igual ao operando B (que contém o imediato da instrução). O restante das operação é atribuída ao campo funct3 e funct, sendo que o seu valor para instruções branch não correspondem exatamente ao mesmo do utilizado para as instruções SLT[U] (dos quais o resultado é aproveitado). Assim se torna necessário dois valores diferentes para a mesma ação de "ALU_op", um que indique o uso do campo funct3 com decodificação específica para instruções branch, e outro para o restante das operações possíveis.

Tabela 3.10 – Codificação do sinal de controle "ALU_op", aonde o valor 11 é utilizado unicamente para a instrução LUI, e 10 para instruções do tipo branch.

ALU_op	Ação
00	adição
01	funct3 e funct
10	funct3-branch
11	LUI

Fonte: Autor.

3.3 MEMÓRIA

A memória foi projetada em duas etapas, inicialmente com leitura assíncrono e escrita síncrona, devido a sua simplicidade para ser descrita em VHDL, e uso em simulações. No momento em que se necessitou de uma memória maior, a fim de não consumir toda a lógica reconfigurável da FPGA, se projeto a memória por blocos de RAM, que possuem resposta síncrona.

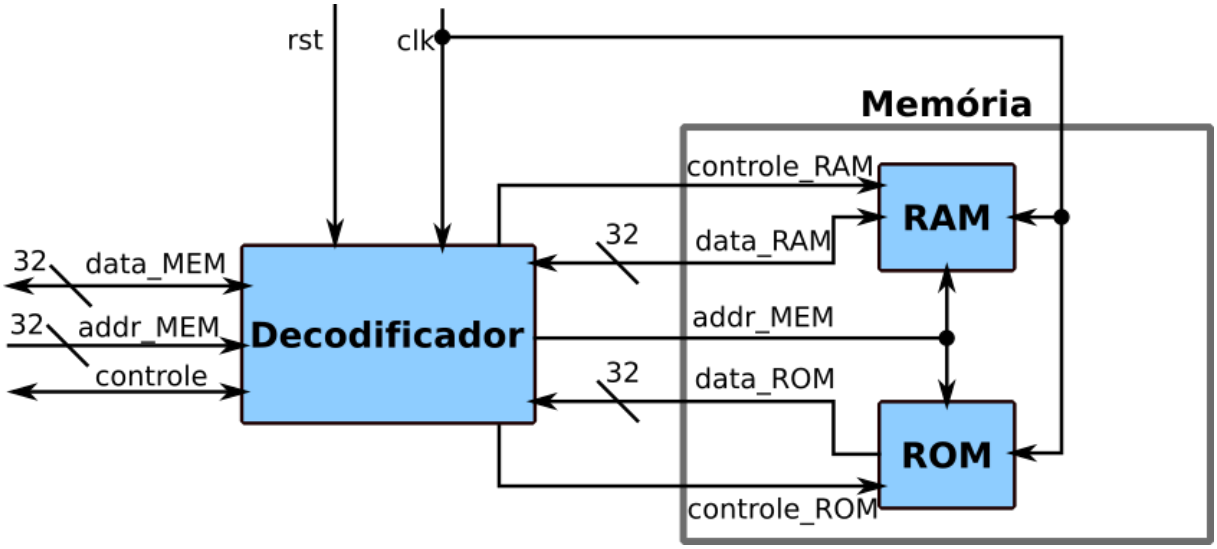
No núcleo multicíclico se tem o uso apenas das instruções RVC (extensão C), e por consequência, a presença apenas de instruções LW e SW para acesso a memória, ou seja, acesso alinhado a memória. Logo, apesar de a ISA base do RISC-V suportar tanto acessos alinhados quanto desalinhados a memória de dados, sendo uma opção de compilação disponibilizada pelas ferramentas do compilador GCC, restringiu-se o suporte apenas a acessos alinhados. Já em relação a busca de instruções, temos dois casos, primeiro para o núcleo multicíclico, aonde o tamanho da instrução é fixo em 16 bits, logo cada instrução deve estar alinhada em 2 bytes. Porém no núcleo com pipeline, o tamanho varia entre 32 bits e 16 bits, relaxando-se assim os acessos, não obtendo exceções de acesso desalinhado.

Em sequência demonstra-se um código que exemplifica o relaxamento, por parte do software, em relação ao acesso desalinhado da RAM, aonde as instruções com o prefixo "c." pertencem a extensão C do RISC-V. Percebe-se que após ser alocada uma instrução de 16 bits para o endereço 4, a próxima a ser executada possui 32 bits de comprimento, que ocupa os endereços de 6 a 9, caracterizando um acesso desalinhado para dados de 32 bits.

```
Disassembly of section .text:

00000000 <main>:
0: 4781 c.li x15,0
2: 4701 c.li  x14,0
4: 76c1 c.lui x13,0xffff0
6: 02b05063 bge x0,x11,26 <main+0x26>
```

A memória foi divida em duas, ROM e RAM, sendo seus acessos controlados pelo decodificador, uma visão de topo é apresentada na Figura 3.22.

Figura 3.22 – Visão de topo da memória, composta pela ROM e RAM, com acesso controlado pelo decodificador.



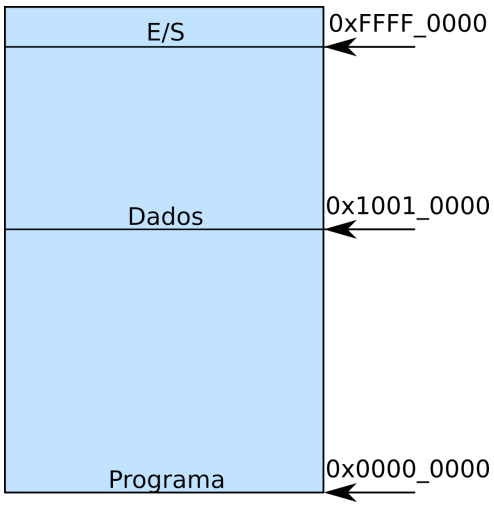
Fonte: Autor.

3.3.1 Mapeamento

Para executar um dado programa no núcleo se necessitou de uma memória capaz de armazenar o programa e os dados. Utilizou-se um layout básico contendo as seções de programa, dados, e por fim a seção dos dispositivos E/S, obtendo-se assim o ambiente mínimo para funcionamento do núcleo. Considerando dados de 32 bits provenientes da ISA utilizada, se tem a capacidade de endereçamento de 4 GBytes, divididos entre as três seções, a divisão da memória em três seções básicas pode ser visualizada na Figura 3.23. Devido a utilização do núcleo para a execução de programas sem a necessidade de retorno a um sistema operacional, se descarta espaços reservados para o mesmo na memória, pois não são utilizados com o circuito atual.

Com a memória limitada do dispositivo FPGA, definiu-se limites dentro do mapeamento para cada uma das seções a serem utilizadas, como se visualiza na Figura 3.24, que possuí duas seções de 16 kBytes, e uma seção para dispositivos E/S, que se resumem a dois, abordados na seção 3.4. Na seção programa se tem o armazenamento do código a ser executado dentro do segmento de texto, e de dados do tipo constate no segmento rodata (read-only data). Para efetivar a inicialização do programa no símbolo "main" de códigos escritos em C, se escreveu uma rotina em assembly, e a alocou em uma seção

Figura 3.23 – Mapeamento de memória utilizado, com as seções programa, que contém o código a ser executado, seção de dados, e a seção dos dispositivos de entrada e saída mapeados diretamente. As setas indicam o endereço inicial de cada seção.



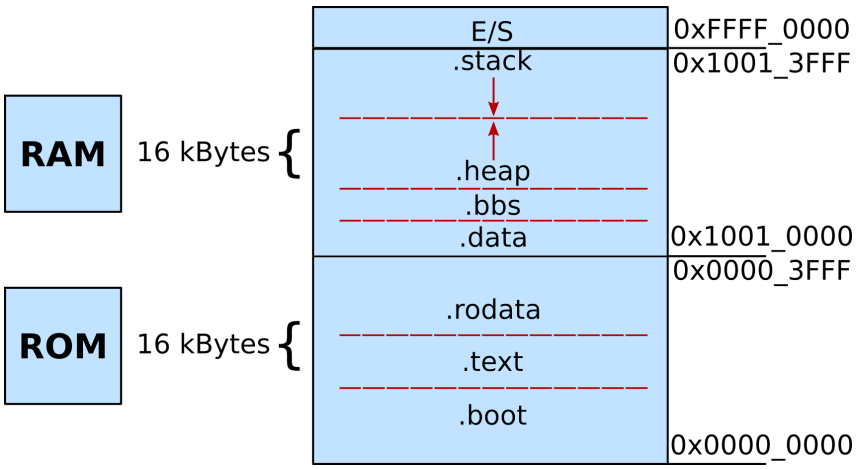
Fonte: Autor.

de boot, que começa no endereço 0 da memória. Sua execução consiste em inicializar o endereço do stack pointer (SP), que na ABI do RISC-V corresponde ao registrador x2, e desviar para a função principal em sequência. A rotina de boot é destinada ao núcleo pipeline, visto que o núcleo multicíclico não executa instruções da base E, e não é possível limitar o compilador ao uso de instruções da extensão C apenas, assim não é possível compilar códigos em C para o mesmo. Confere-se a seguir a simples rotina de atribuição do SP e desvio para o símbolo "main":

```
.section .boot
boot_rv32ec:
# set stack pointer value, x2 = stack pointer = sp
# lui sp,0x00008
# addi sp,sp,-1
li sp,0x00007FFF # pseudo-instruction 'li'

# jump to function "main"
j main
```

Para a seção de dados se obtém uma divisão de quatro segmentos, data que possuí os dados inicializados, bbs com os dados não inicializados, seguido pelo segmento heap e stack, que crescem em sentidos opostos. Para cada seção se utilizou diferentes memórias, adicionando-se um decodificador do endereço de entrada para habilitar a correspondente, incluindo os dispositivos E/S, pois se tem que eles estão mapeados diretamente na memória, reduzindo assim o espaço total de memória disponível para programas.

Figura 3.24 – Mapeamento da memória utilizado na prática devido a quantidade de memória disponível no dispositivo FPGA Spartan-6 com uso de BRAMs, indicando os endereços iniciais e finais de cada seção, compostas pelos componentes ROM e RAM.



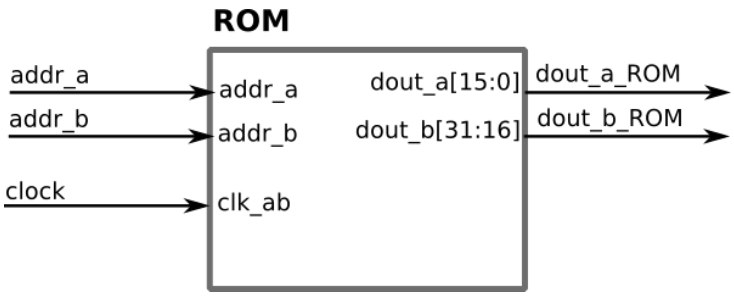
Fonte: Autor.

Para a implementação se definiu um número fixo de blocos de RAM a serem utilizados como memória de programa e dados. Com um total de 32 BRAM disponíveis na FPGA adotada para o projeto, foram utilizadas 16, sendo 8 para a memória de programa, e as 8 restantes para dados. Cada bloco contendo um total de 2 kBytes, logo se obteve 16 kBytes de memória de programa e também de dados, para a execução do núcleo. Como característica síncrona dos blocos de RAM da FPGA spartan-6, tanto a leitura quanto a escrita são realizadas com uma latência de um ciclo de relógio, ou mais caso se escolhesse aumentar o número de registradores de saída da memória (pipeline), porém optou-se por manter-se apenas o primeiro registrador.

3.3.2 Memória de Programa - ROM

A ROM foi projetada com duas portas de leitura, com largura de 16 bits, com suas respectivas saídas, A e B, com dados também de 16 bits, e sem porta de escrita. Apesar de as instruções para o núcleo multicíclico serem do tamanho exato de uma única porta da ROM, se tem a necessidade de duas. Isso ocorre pela presença de dados de 32 bits do segmento rodata, que são armazenados na mesma. A Figura 3.25 apresenta as portas de entrada e saída do componente, aonde se indica pelos sufixos "a" e "b" a qual pertence, ou quando a porta é compartilhada por ambas, como no caso da entrada de clock. Se indica também qual porta compõe a parcela mais significativa de um dado lido da ROM, no caso, a porta B.

Figura 3.25 – Portas de entrada e saída da ROM, projetada como um bloco de RAM.



Fonte: Autor.

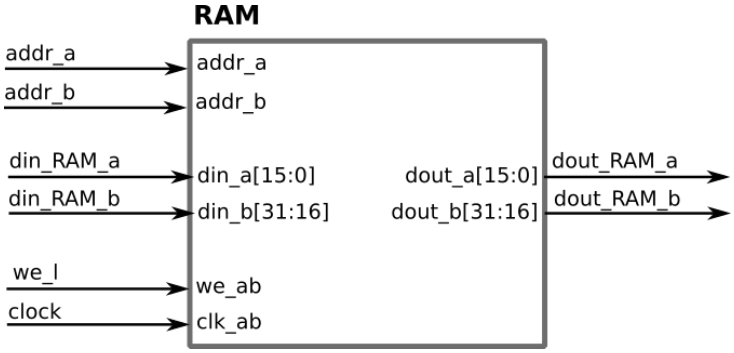
3.3.3 Memória de Dados - RAM

Para a memória RAM obteve-se dois casos, um considerando o núcleo multicíclico, com acesso a palavras apenas, e outro no pipeline, em que é possível acessar byte ou half-word.

3.3.3.1 RAM - Multicíclico

No multicíclico a RAM possuí duas portas de 16 bits, sem habilitação de byte, ou seja, qualquer escrita realizada, irá necessariamente escrever 32 bits, de acordo com o sinal compartilhado por ambas as portas, "weab". É indicado na Figura 3.26 suas portas de entrada e saída, diferenciando as que pertencem a porta A, B, ou ambas, como o sinal de habilitação de escrita e clock.

Figura 3.26 – Portas de entrada e saída da RAM, projetada como um bloco de RAM.



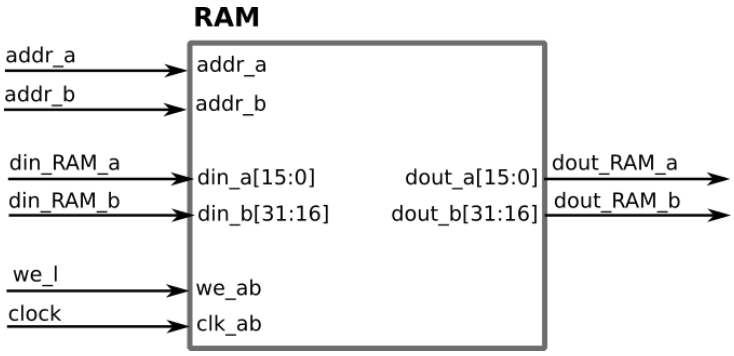
Fonte: Autor.

3.3.3.2 RAM - Pipeline

Para o núcleo com a técnica de pipeline se tem instruções da base E como SB/SH, resultando então em uma RAM com duas portas de 16 bits, como já obtido no uso do núcleo multicíclico, porém adicionando-se a lógica para habilitação de escrita por byte. A Figura 3.27 contém as portas de entrada e saída da RAM utilizada em conjunto ao núcleo pipeline. Se "we_a(0)" estiver em nível 1 e "we_a(1)" e nível 0, a escrita será habilitada apenas para os dados de entrada da porta A, no campo [7:0], deixando o byte mais significativo inalterado. A habilitação por escrita de byte é utilizada apenas na porta A, quando se escreve na B, então a escrita é necessariamente de uma palavra.

Supondo um acesso no endereço de valor 1, considerando que o núcleo opera com endereçamento a byte, e um store byte para esse mesmo endereço. Essa operação não seria possível, pois o endereçamento do bloco de RAM está por palavras de 16 bits, logo, iria escrever no endereço 2. Uma solução envolveria ignorar o LSB do endereço de acesso, e utiliza-lo para multiplexar os 2 bits de "we_a" entre si, quando necessário. Sendo assim, o LSB das portas de endereço está fixo em nível 0.

Figura 3.27 – Portas de entrada e saída da RAM, projetada como um bloco de RAM.



Fonte: Autor.

3.3.4 Decodificador

O decodificador recebe o endereço de acesso e o compara com o inicio das seções para as quais cada memória corresponde, habilitando a correspondente pelos sinais de controle "cs_ROM" e "cs_RAM". O dispositivo que estiver desabilitado irá ter o seu barramento colocado em alta impedância, evitando conflitos de dados, o mesmo serve para os componentes IO, pois quando um endereço é de sua seção, o sinal de habilitação do mesmo é proveniente da unidade de decodificação da memória, visto que os dispositivos são mapeados diretamente. Um dos sinais de entrada do decodificador, que vem diretamente do núcleo, é o de validade do endereço, quando ativo, por mais que o endereço

esteja contido na região da ROM, por exemplo, o memória não será habilitada. São utilizados somente os bits [31:16] do endereço de entrada do decodificador para habilitação das seções.

Quando um sinal de habilitação da memória (ROM ou RAM) está ativo, inicia-se a contagem de latência de acesso internamente no decodificador, para indicar ao núcleo quando o dado já está disponível no barramento. Como definiu-se um único registrador de pipeline para os blocos de RAM, a contagem é de apenas um ciclo de relógio. Sendo que, no caso do endereço não ser válido, o chip de memória correspondente não somente não será habilitado, assim como a contagem não irá iniciar. Evitando assim que em um ciclo posterior o decodificador indique ao núcleo acidentalmente que o dado no barramento é útil.

3.3.4.1 Decodificador - Núcleo Multicíclico

A Figura 3.28 contém o esquemático interno do decodificador utilizado em conjunto ao núcleo multicíclico, aonde cinco blocos internos são apresentados. Três deles se resumem aos comparadores, para definirem o estado dos sinais de habilitação da memória, que se provarem que a condição é correta, e contanto que o sinal de validade do endereço esteja ativo, irá habilita a memória correspondente. A Tabela 3.11 possui a lógica dos comparadores, aonde se evidência que os sinais de habilitação são considerados ativos em 0.

Tabela 3.11 – Habilitação das seções da memória de acordo com o endereço de entrada.

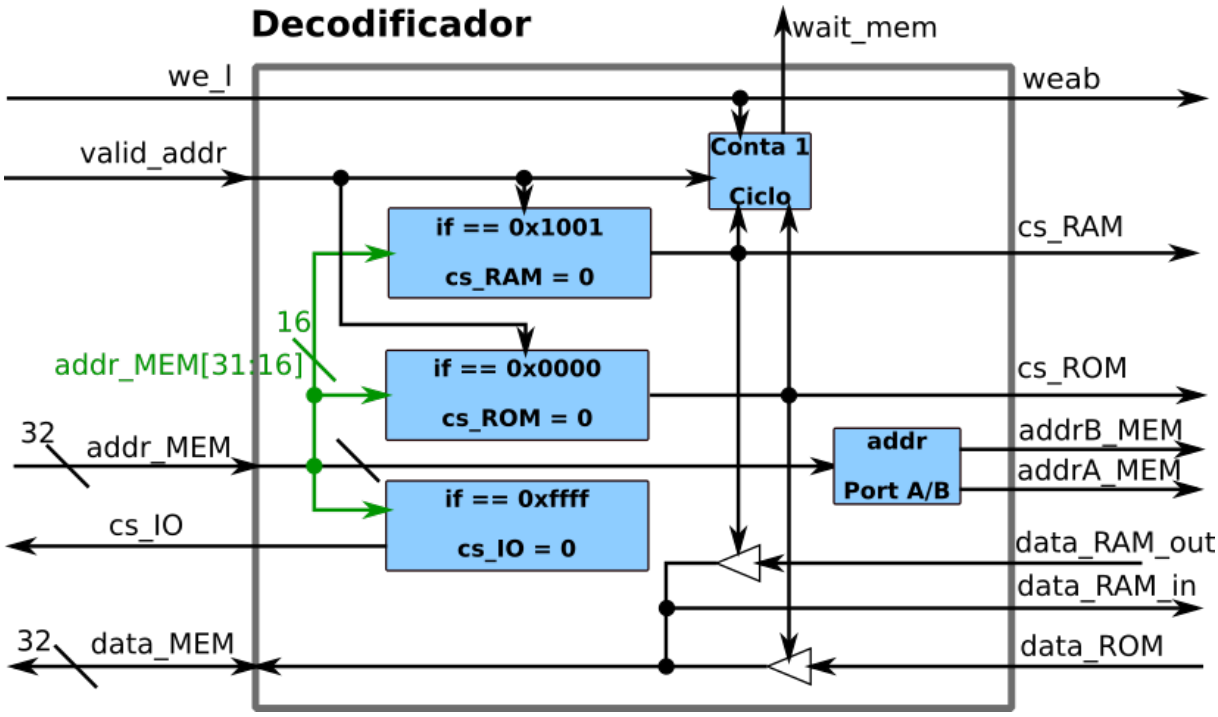
valid_addr	addr_MEM[31:16]	cs_ROM	cs_RAM	cs_IO
1	0xffff	1	1	0
1	0x1001	1	0	1
1	0x000a	0	1	1
0	—	1	1	1

Fonte: Autor.

O endereço de acesso a memória é o mesmo para ambas, ROM e RAM, sendo ele divido em dois dentro do bloco, um para cada porta de acesso dos blocos de RAM. O da porta A corresponde ao endereço inalterado, já o utilizado na porta B é o seu offset por 2, totalizando assim o acesso a 4 bytes. O barramento de dados principal é conectado por meio de portas tri-state, colocando em nível de alta impedância a memória não habilitada para evitar conflitos.

Por fim, o bloco "Conta 1 ciclo" se refere ao contador de latência, indicando quando o dado no barramento está pronto para ser lido pelo núcleo. A contagem só se inicializa quando um dos chips está habilitado, e o endereço é válido, porém ainda é necessário uma

Figura 3.28 – Esquemático interno do decodificador utilizado em conjunto com o núcleo multicíclico.



Fonte: Autor.

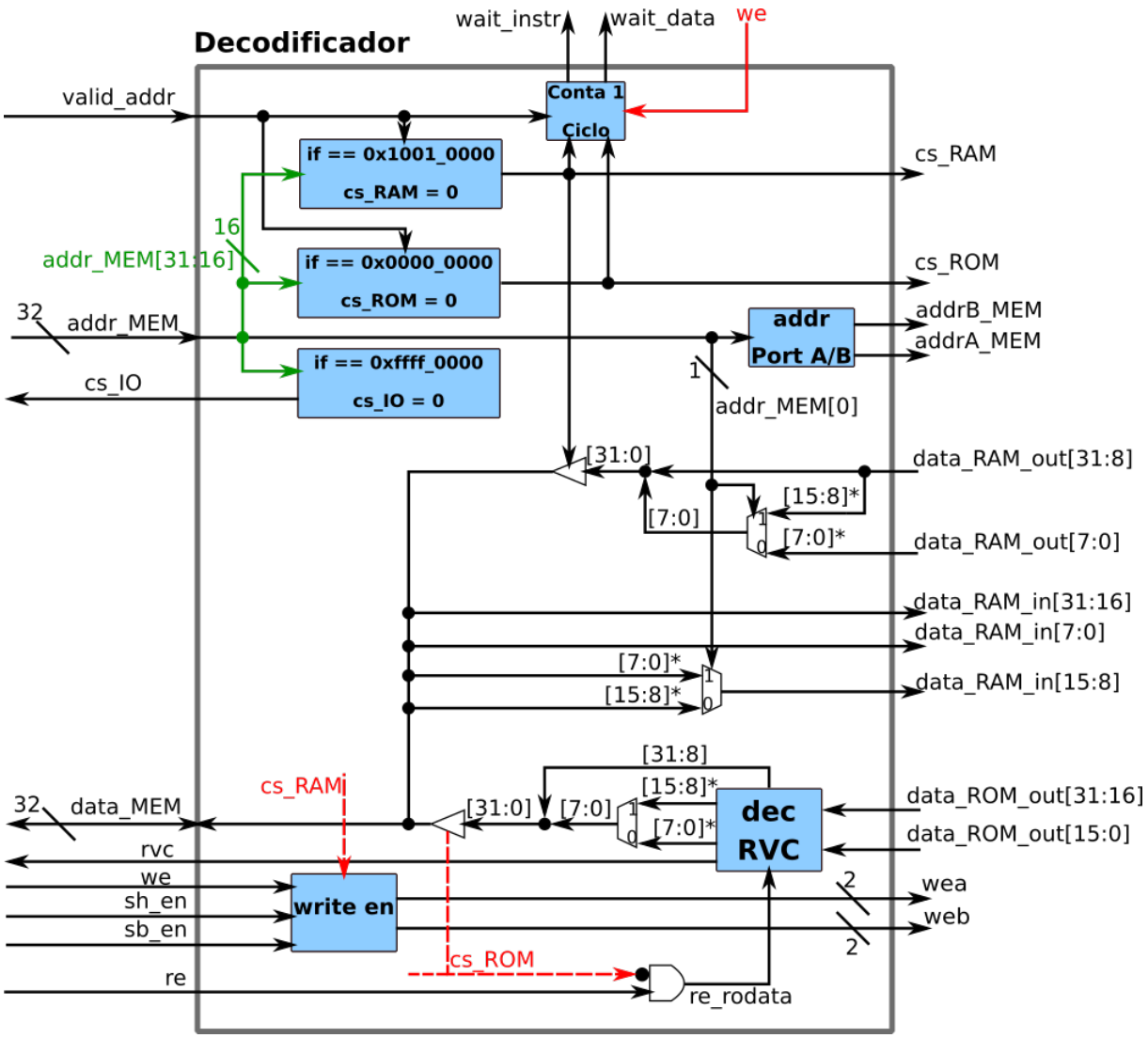
lógica extra, formada pelo sinal de escrita. Quando a escrita está habilitada, e se tem uma acesso a memória RAM, não se ativa a espera por um ciclo de relógio do núcleo (indicada pelo sinal wait_mem), pois a mesma é desnecessária nesta operação, visto que a escrita irá se concretizar na primeira borda de subida de forma síncrona. A habilitação de escrita, renomeada de "we_l" para "wea" dentro do decodificador para fazer referência as portas A e B da BRAM, é um único sinal compartilhado para ambas.

3.3.5 Decodificador - Núcleo Pipeline

O decodificador utilizado em conjunto com o núcleo pipeline, da mesma forma que o multicíclico, utiliza três comparadores para habilitar as seções de memória correspondentes, de acordo com o endereço de acesso, ignorando-se caso o endereço não seja válido. A função de contagem de latência de acesso possuí duas saídas, uma referente ao tipo de acesso realizado, para leitura de instruções, ou de dados (que pode estar na RAM, ou na ROM, seção rodata). Essa separação é essencial, pois no pipeline os dois tipos de finalidades de acesso são realizados em estágios diferentes, e suas ocorrências acabam por resultar em um controle diferenciado, travando registradores diferentes. Para evitar que o núcleo fique um ciclo a mais ocioso esperando uma operação de escrita, o

sinal "we" (write enable) é conectado ao contador, de forma a desabilita-lo, e indicar ao núcleo para não realizar nenhum stall. A Figura 3.29 demonstra o esquemático interno do decodificador.

Figura 3.29 – Esquemático interno do decodificador da memória utilizado em conjunto ao núcleo pipeline. Os sinais we, cs_RAM, e cs_ROM foram multiplicados internamente no diagrama, e destacados, para evitar a poluição da mesma.



Fonte: Autor.

O bloco "addr Port A/B" tem por finalidade dividir o endereço de acesso a memória em dois, um para a porta A, que é o mesmo de entrada do decodificador, e outro para a porta B, sendo o seu offset por 2. Assim se totaliza o acesso aos 4 bytes desejados, sendo que o endereço é o mesmo para ambas as memórias. Um destaque se da por conta da função de escrita, que possuí habilitação de escrita por byte, de acordo com os sinais de controle de entrada "sb_en" e "sh_en". Na Tabela 3.12 visualiza-se a lógica da habilitação de escrita, aonde divide-se o sinal de saída em dois, "wea" para a porta A, e "web" para a

porta B. Operações de escrita a byte e meia palavra ocorrem sempre na porta A, logo, a porta B não utiliza da função byte enable dos blocos de RAM, e seu sinal é de 1 bit.

Para a porta A, a habilitação de escrita possuí largura de 2 bits, sendo o LSB para os byte menos significativo. Observa-se que, para a execução de armazenamento de um único byte no campo [15:8] necessita-se ainda de lógica adicional, pois as memórias armazenam 16 bits por endereço. A determinação é realizada pelo LSB do endereço do decodificador, quando o mesmo está em nível 0, o byte a ser escrito é o inferior. Para nível 1, se escreve no byte superior da porta, e através de um multiplexador (controlado pelo LSB do endereço) se transfere o byte a ser escrito, no campo [7:0] do barramento de dados.

Para a leitura se tem a mesma lógica de uso do LSB de endereçamento, para acesso ao byte superior da porta A. Quando o mesmo for de nível 1, se multiplexa o campo [15:8] da saída da memória, para o desejado [7:0]. O resto da lógica de leitura de byte/half-word é realizada dentro do núcleo, como extensão do sinal, ou concatenação com 0. Em suma, é essencial que tanto na operação de escrita, quanto de leitura, o byte a ser manipulado esteja na posição menos significativa. Ambas as memórias, ROM e RAM, contam com esse multiplexador para leitura do byte superior, sendo o da ROM localizado após o bloco decodificador RVC.

Tabela 3.12 – Decodificação da habilitação de escrita com byte enable do bloco de RAM.

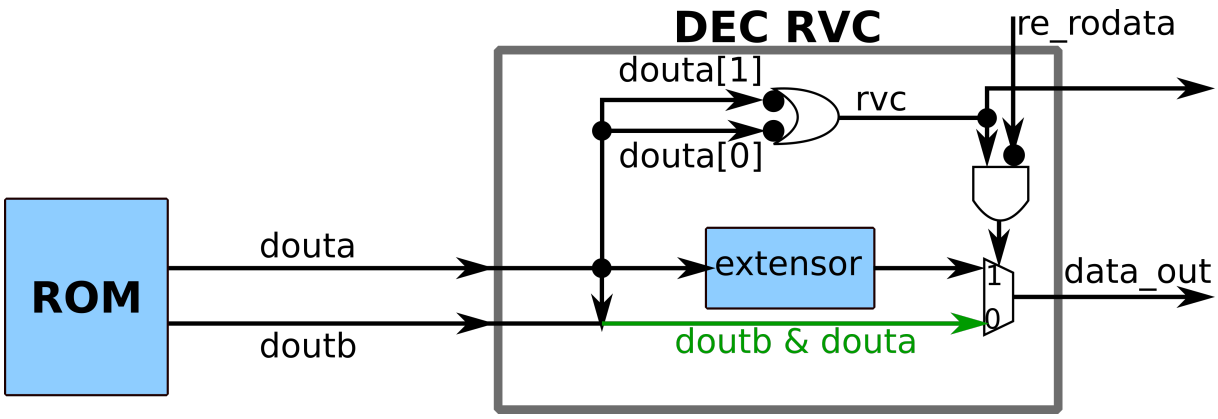
cs_RAM	we	sb_en	sh_en	wea[1:0]	web[0]
0	1	0	0	11	1
0	1	1	0	01	0
0	1	0	1	11	0
x	0	x	x	00	0
1	x	x	x	00	0

Fonte: Autor.

O decodificador RVC está presente apenas para uso com núcleo pipeline, tendo a função de estender as instruções de 16 bits da extensão C, para sua correspondente da base E de 32 bits, durante a etapa de busca da mesma. A Figura 3.30 contém o esquemático interno do decodificador RVC. Sua entrada de dados consiste na leitura da ROM (memória de programa) e um único sinal de controle nomeado de "re_rodata" (read enable rodata). O sinal de controle serve para evitar que o decodificador RVC acidentalmente modifique dados constantes (que são armazenados na ROM) por considera-los como instruções, multiplexando então a entrada diretamente para a saída.

A verificação do tamanho da instrução é realizada pelos dois LSB do dado de entrada, que correspondem a uma parcela do campo opcode da instrução da base E, que quando possuírem qualquer bit diferente do nível lógico 1, indica que é uma instrução RVC. Conforme a Tabela 3.13, dentre os três possíveis valores em que se obtém uma instrução

Figura 3.30 – Decodificador de instruções da extensão C, localizado na saída da ROM.



Fonte: Autor.

RVC, três quadrantes (seções) são obtidas, em que instruções com características semelhantes são agrupadas. Essa característica da extensão C auxilia na decodificação, por exemplo, o imediato terá o sinal estendido apenas no 2º quadrante (opcode[1:0] = 01), e para o restante utiliza-se da concatenação com 0.

Tabela 3.13 – Determinação do tamanho da instrução pelos dois LSB do dado lido da memória de programa.

opcode[1:0]	Instrução (conjunto - tamanho)	sinal rvc
00	ext. C - 16 bits, 1º Quadrante	1
01	ext. C - 16 bits, 2º Quadrante	1
10	ext. C - 16 bits, 3º Quadrante	1
11	base E - 32 bits	0

Fonte: Autor.

Uma saída é necessária para indicar ao núcleo quando uma instrução é de 16 bits, para que o mesmo incremente o PC por 2, e não 4, sendo esse sinal o "rvc", que ativo em nível 1 indica uso da extensão C, do contrário, base E. O sinal "re_rodata" é gerado internamente no decodificador quando se tem um acesso na ROM (cs_ROM = 0), e o sinal de habilitação de leitura da memória está ativo (re = 1), considerado ativo em 1.

3.4 DISPOSITIVOS E/S

Dois componentes E/S foram projetados, uma unidade de comunicação para troca de dados entre o núcleo e o computador pessoal utilizado, no caso uma UART, e um temporizador para contar intervalos de tempo. O núcleo não foi projetado de forma a

receber interrupções, ou seja, quando se tem a necessidade do uso de um determinado dispositivo externo, é utilizada a técnica de polling, verificando-se continuamente o estado atual dos dispositivos. Fator este que diminuí drasticamente o seu desempenho em uma aplicação prática. A UART e seu controle utilizaram os primeiros dois endereços da seção E/S, com um registrador de controle de 32 bits, indicando quando um dado está pronto para escrita ou leitura, e outro endereço de dados. O buffer de dados da UART é de 8 bits, enviando sempre os bits menos significativos em uma instrução store word, em uma leitura se te a concatenação com 0.

O temporizador ocupa um único endereço, utilizado para leitura e escrita, o sinal de relógio de entrada é o mesmo do núcleo e esta constantemente incrementando o valor de contagem a cada borda de subida quando o sistema é inicializado após a ação do sinal de reset. A Tabela 3.14 demonstra os endereços de E/S ocupados, com suas finalidades de acesso.

Tabela 3.14 – Endereços da seção E/S ocupados pela UART e pelo temporizador, indicando-se quais operações de acesso estão disponíveis.

Dispositivo E/S	Endereço	Função
UART Registrador de Controle	0xFFFF_0000	Leitura
UART Registrador de Dados	0xFFFF_0004	Leitura/Escrita
Temporizador	0xFFFF_0008	Leitura/Escrita

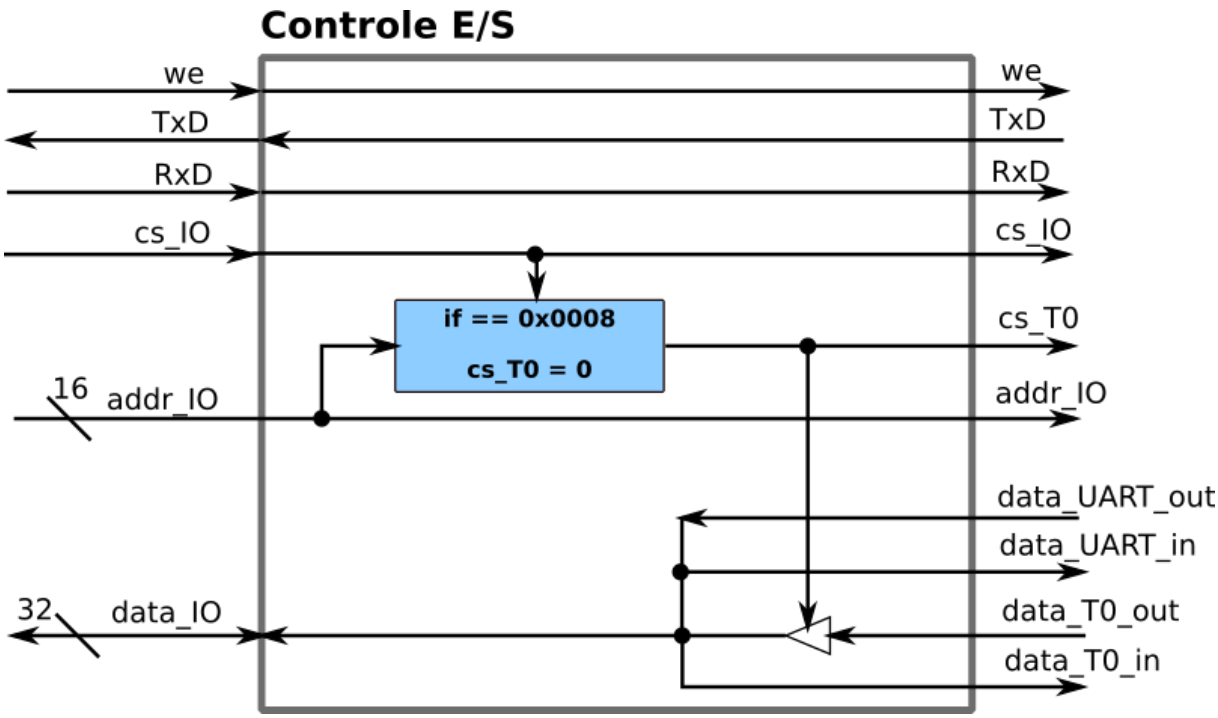
Fonte: Autor.

Para controlar os dispositivos se utiliza de uma unidade de controle, a fim de decodificar seus acessos, conforme demonstra-se na Figura 3.31. Sua entrada "cs_IO" em conjunto com o endereço de acesso, determina qual dispositivo será habilitado, se o nível de "cs_IO" for 1 (desabilitado), ignora-se qualquer acesso. O sinal de habilitação dos dispositivos ("cs_IO") é proveniente da unidade decodificadora da memória, discutida na Seção 3.3, sendo gerada a partir dos 16 MSB do endereço, logo só necessita-se dos 16 LSB restantes para uso nos dispositivos E/S. Visto também que em toda a seção de dispositivos E/S da memória, os 4 dígitos hexadecimais mais significativos correspondem a F.

Os sinais de habilitação são todos ativos em 0, sendo o do timer, "cs_T0", gerado pela comparação dos 16 LSB do endereço. O mesmo sinal habilita o buffer tri-state da saída do temporizador, caso seja operação de leitura. A UART possuí internamente a lógica de comparação aos endereço de acesso, logo os sinais necessários apenas são repassados para a unidade. A unidade UART também realiza a lógica de alta impedância do barramento de dados, para não causar conflitos entre suas saídas com o barramento interno do controle de dispositivos E/S. Outros dos sinais de UART transitam internamente pelo controle, sem alterações, sendo "TxD" e "RxD", que correspondem a transmissão e recepção de dados serial. Por fim, está presente o sinal de habilitação de escrita, "we", ativo

em nível lógico 0, e compartilhado por ambos os dispositivos. Ocultou-se na representação de blocos, mas tanto a UART quanto o temporizador ainda recebem como entrada o sinal de clock do sistema, e também de reset.

Figura 3.31 – Controle dos dispositivos E/S.



Fonte: Autor.

3.4.1 UART

Não se utiliza todos os 32 bits (4 endereços) ocupados pelo acesso a ambos os registradores da UART, porém no caso de mover-se o registrador de dados para o endereços "0xFFFF_0001", por exemplo, necessitaríamos da instrução de acesso a byte (load byte/store byte) para o mesmo. A extensão C não conta com instruções de acesso a byte e half-word, apenas load/store word. Logo o uso do registrador de dados da UART iria gerar um de endereço desalinhado para o núcleo multicíclico, indesejado no projeto.

No total, utilizaram-se apenas os 4 LSB do registrador de controle/status da UART, de acordo com a Tabela 3.15.

• UART Tx Ready: em nível 1 indica que a UART está pronta para enviar o próximo dado;

• UART Tx Write: em nível 1 indica que a UART recebeu um novo dado que está pronto para ser transmitido;

Tabela 3.15 – Campos do registrador de controle da UART.

bit	Sinal
(0)	UART Tx Ready
(1)	UART Tx Write
(2)	UART Rx New Data
(3)	UART Rx Read

Fonte: Autor.

• UART Rx New Data: em nível 1 indica que a UART recebeu um novo dado que está pronto para ser lido;

• UART Rx Read: em nível 1 indica que a UART está pronta para receber o próximo dado.

Os bits "UART Tx Write", e "UART Rx Read" são utilizados como flag pela UART para habilitar ou desabilitar a comunicação temporariamente enquanto se espera uma transmissão ser finalizada. Sendo os bits "UART Tx Ready" e "UART Rx New Data", utilizados por parte do programador para verificar por polling quando o dispositivo está pronto para ser utilizado novamente.

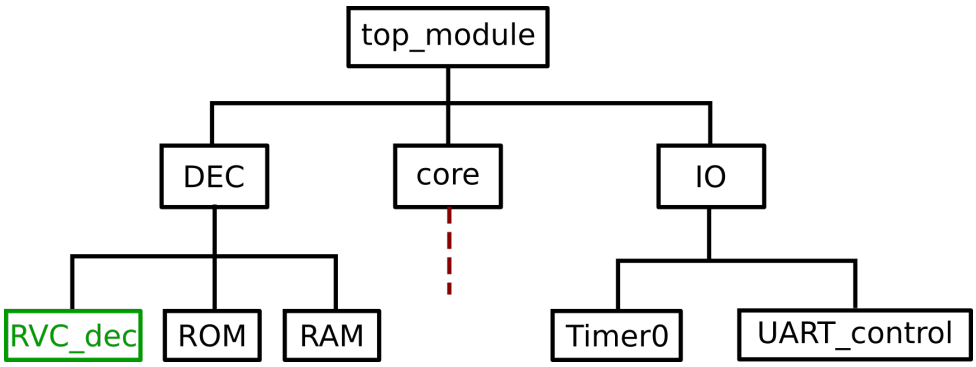
4 RESULTADOS E DISCUSSÃO

4.1 CONSIDERAÇÕES GERAIS

Este capítulo relata os resultados obtidos do projeto, dois núcleos são apresentados em termos de consumo de área lógica da FPGA Spartan-6, e também da frequência de operação máxima dos mesmos, de acordo com os dados coletados no software Xilinx ISE Design Suite 14.7. A ocupação total de lógica do dispositivo FPGA é uma consequência direta de decisões relacionadas a forma como foram projetados os hardwares, por exemplo, nos prós e contras em utilizar um bloco de RAM para a implementação do banco de registradores. Diagramas de tempo são apresentados com simulações especificas, de ambos os núcleos multicíclico e com pipeline. Também se demonstra a organização da memória utilizada para a execução de programas escritos em assembly, por meio do script de link criado a partir do layout desejado para a memória.

A descrição em VHDL de todo o sistema, incluindo memória e dispositivos E/S, foi organizada conforme a hierarquia da Figura 4.1, aonde a unidade "RVC dec" está presente apenas na implementação com o núcleo pipeline. Os elementos internos do núcleo estão ocultados, visto que variam para cada uma das duas implementações.

Figura 4.1 – Hierarquia da descrição em VHDL do sistema.



Fonte: Próprio Autor.

4.2 NÚCLEO

Ambos os núcleos descritos na Seção 3.2, foram implementados com a ferramenta de análise das restrições de tempo, OFFSET IN, OFFSET OUT e PERIOD, para 10 ns, ou seja, para se utilizar do clock disponível na placa Nexys 3 de 100 MHz. A Seção 4.2.1

discute os resultados relacionados ao banco de registradores, enquanto que nas Seções 4.2.2 e 4.2.3 demonstra-se os resultados dos núcleos multicíclico e pipeline, assim como simulações das suas classes de instruções.

4.2.1 Banco de Registradores

O banco de registradores é um dos componentes de destaque do núcleo, visto que a base E do RISC-V implementada possui como característica justamente a sua redução de 32 registradores de uso geral, para apenas 16. A implementação dos elementos de memória (16 registradores) é da mesma forma em ambos os núcleos, havendo diferenças apenas em questões como o forward interno do pipeline, que não é necessário no núcleo multicíclico. O uso da base E tem por objetivo a obtenção de um núcleo reduzido, mais encontrado em sistemas embarcados, que levou a escolha de implementar o banco de registradores com uso das LUTs da FPGA Spartan-6.

A opção mais direta, sendo que o banco consiste em uma pequena memória, seria a da utilização dos blocos de RAM, dessa forma se reduziria a lógica da FPGA utilizada. Porém ao utilizar-se um bloco de RAM unicamente para a implementação de uma memória 32x16, como resultado de que as BRAMs podem ser configuradas com profundidade e comprimento 32x512 (XILINX, 2011b, pág. 10), a maior parte de seu armazenamento seria desperdiçado. Para se utilizar esse espaço de blocos de RAM desocupados seria necessário a adição de lógicas para controlar as saídas e entradas, assim como permissões de escrita por byte.

Também considerou-se o fato de que os blocos de RAM estão posicionados de forma fixa internamente no CI, dessa forma já se obtém uma limitação quanto a otimização durante a etapa de síntese e implementação da descrição em VHDL, realizada pelo software ISE Design. Outro ponto fundamental que levou a escolha de implementar o banco de registradores por meio de LUT está no número de portas dos blocos de RAM da Spartan-6. O número máximo de portas é de duas, sendo uma para de leitura/escrita no endereço A, e o outra para o endereço B (escrita e leitura também). Assim não se tem porta de acesso para todos os três endereços desejados, que são rs1, rs2 e rd, dois de leitura e um de escrita. Uma abordagem direta iria envolver o uso de uma BRAM extra apenas para a porta de escrita, aumentando o espaço de memória desperdiçado internamente na FPGA. Somando-se a isso, se obtém um aumento na complexidade da lógica de forward no banco de registradores realizando leituras síncronas.

4.2.2 Núcleo Multicíclico - RVC

A hierarquia da descrição em VHDL pode ser visualizada na Figura 4.2. Os resultados obtidos de consumo lógico da FPGA, considerando implementações unicamente do núcleo e também do sistema completo (memória e dispositivos E/S), podem ser visualizados na Tabela 4.1, aonde também se indica a frequência máxima estimada pela ferramenta de análise temporal do software ISE Design 14.7, alcançando os 100 MHz mínimos de operação desejado. Percebe-se que na implementação do núcleo sozinho se obteve uma velocidade de operação menor em comparação ao processador, fato atribuído ao grande número de conexões extras nas portas da FPGA. Quando se implementa o núcleo apenas, o número de portas de saída e entrada é de 64 a mais que no processador, correspondendo ao barramento de dados e o de endereços, influenciando bastante no desempenho do hardware. Nota-se também o pequeno acréscimo de consumo das LUTs no sistema completo, do núcleo para o processador, 10% e 13%, sendo essa pequena diferença de área devido ao uso de BRAMs para a memória. O sumário completo de utilização dos recursos do dispositivo pode ser consultado nos Apêndices B.1 e B.2, e a descrição final VHDL no Apêndice E.

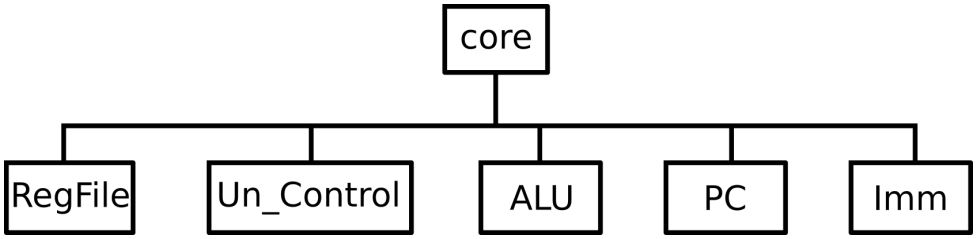
Tabela 4.1 – Resumo de utilização de recursos da FPGA Spartan-6 para a versão do caminho de dados multicíclico, considerando implementações unicamente do núcleo, e também do núcleo com os demais componentes para sua operação (memórias e dispositivos E/S).

Recursos	Disponível	Usado	Utilização (%)
Núcleo			
Número de Slice Registers	18.224	696	03%
Número de Slices LUTs	9.112	961	10%
Número de Slices Ocupados	2.278	429	18%
Blocos de RAM	32	0	00%
Frequência Máxima (MHz)		101,74	
Sistema Completo			
Número de Slice Registers	18.224	842	04%
Número de Slices LUTs	9.112	1.223	13%
Número de Slices Ocupados	2.278	518	22%
Blocos de RAM	32	16	50%
Frequência Máxima (MHz)		104,167	

Fonte: Autor.

A execução das instruções RVC implementadas no núcleo foram divididas de acordo com os possíveis caminhos do diagrama de estados da Figura 3.8 da unidade de controle apresentada na Seção 3.2.1.4. O estado A e B são comuns a todas as instruções executadas, correspondendo as ações de busca da instrução da memória e decodificação da instrução/leitura do banco de registradores, respectivamente. Após isso o diagrama se divide em um total de 10 caminhos possíveis, agrupando instruções com características de

Figura 4.2 – Hierarquia da descrição em VHDL do núcleo multicíclico RVC.



Fonte: Autor.

execução comum no núcleo. O estado D é um ponto de convergência, visto que realiza a escrita no banco de registradores (presente em diversas instruções). A Tabela 4.2 nomeia as classes de instrução de acordo com o rumo tomado pela máquina de estados durante a execução da instrução, a partir do estado B do diagrama.

Tabela 4.2 – Divisão das classes de instruções do núcleo multicíclico com base nos caminhos do diagrama de estados.

Classe	Característica	Estado
Reg-to-Reg	operação reg. com reg.	C -> D
Reg-to-Imm	operação reg. com imm	E -> D
LI/MV	instrução LI e MV	F -> D
ADDI4SPN	instrução ADDI4SPN	G -> D
Desvio cond.	instruções de branch	K
Desvio incond. PC	instruções jump	J
Desvio incond. REG	instruções jump register	L
Load	leitura da memória	G -> I -> D
Store	escrita da memória	G -> H

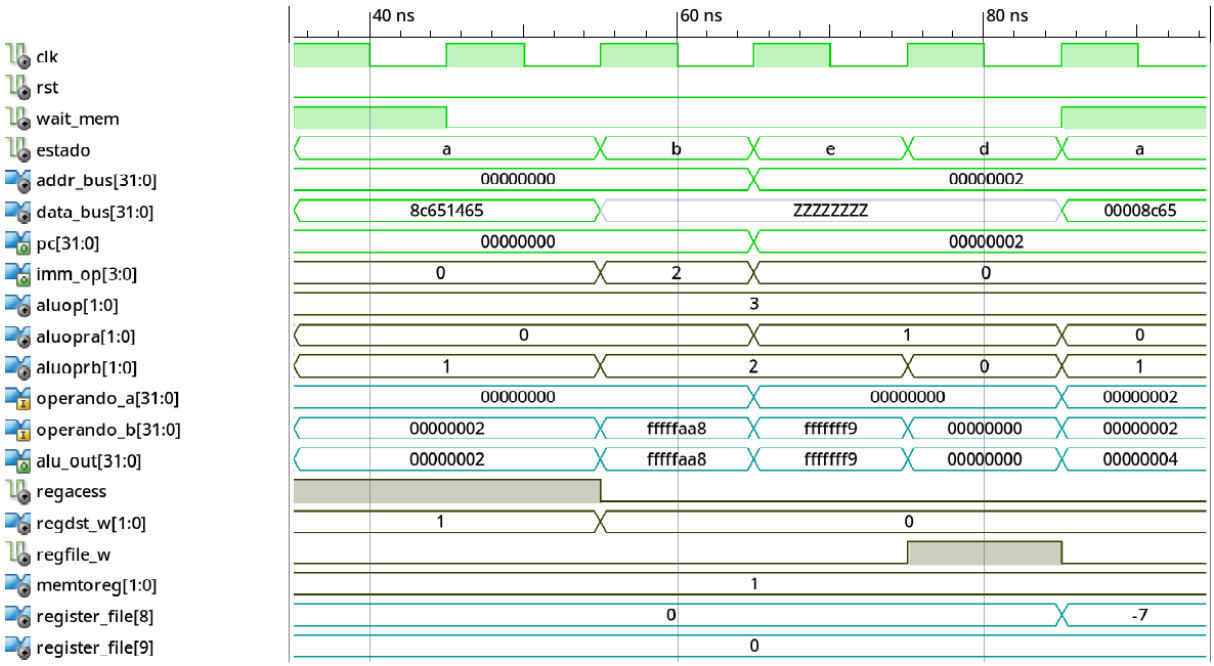
Fonte: Autor.

4.2.2.1 Classe Reg-to-Reg e Reg-to-Imm

Na classe reg-to-imm se tem a execução das instruções: c.addi, c.andi, c.slli, c.srli, c.srai e c.addi16sp. O diagrama de tempo da Figura 4.3 demonstra a execução da instrução c.addi, aonde a mesma tem por objetivo somar o registrador x8 (com valor inicial 0) ao imediato -7, resultando no mesmo valor armazenado em x8. Durante os primeiros dois ciclos a máquina de estados permanece no estado A, indicado pelo sinal "estado", devido a latência de acesso a memória, conforme o sinal "wait_mem". Após isso, no estágio B se realiza a decodificação da instrução, e leitura do banco de registradores.

De acordo com a decodificação a máquina decide o próximo estado, sendo no caso, o E, que realiza a operação de soma do valor contido no registrador x8 (register_file[8]) com

Figura 4.3 – Execução da instrução c.addi da classe reg-to-imm do núcleo multicíclico. Todos os sinais estão representados em hexadecimal, com exceção dos registradores x8 e x9 do banco, sinais "register_file[8]" e "register_file[9]", respectivamente.



Fonte: Autor.

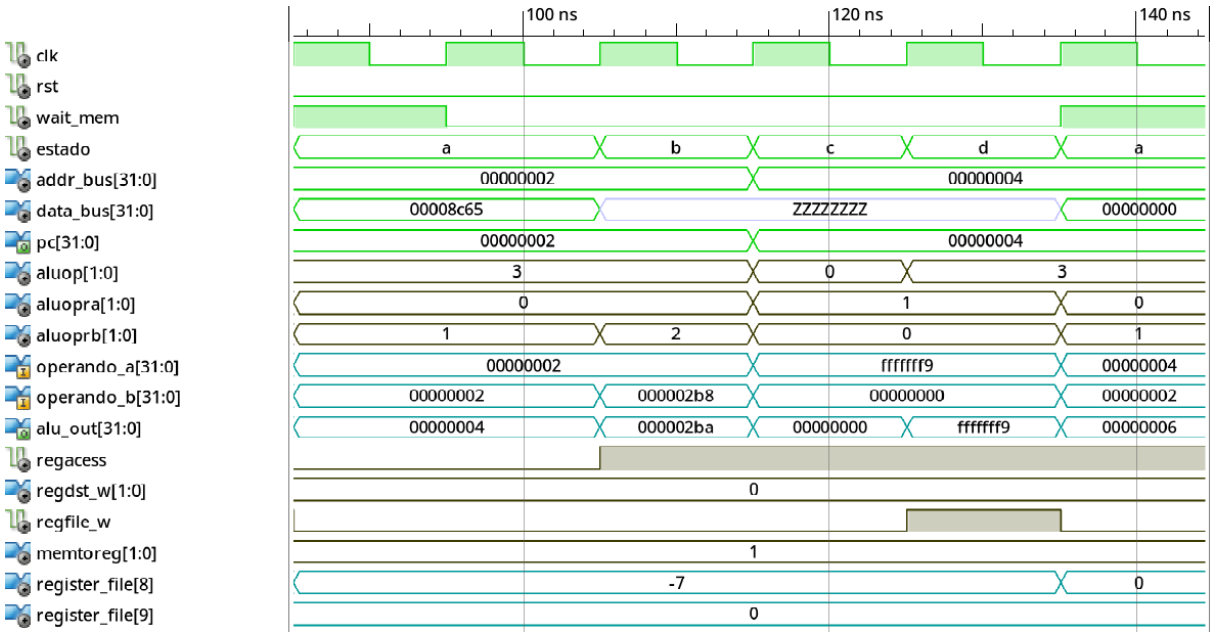
o imediato -7. Os sinais de controle essenciais ao funcionamento da instrução, conforme destacados na Seção 3.2.1.4, estão presentes no diagrama de tempo da Figura 3.8 para inspeção. No ciclo em que a máquina se encontra no estado D se realiza então a escrita no banco, ao colocar em nível 1 (ativo) o sinal de permissão de escrita "regfile_w", visualizando-se no ciclo seguinte após a borda de subida do clock, o valor -7 no registrador x8.

Em sequência a essa instrução reg-to-imm, executou-se uma da classe reg-to-reg, que contém as seguintes possibilidades: c.add, c.sub, c.and, c.or e c.xor. O diagrama de tempo da Figura 4.4 contém a simulação de uma instrução c.and, entre os registradores x8 e x9, armazenando o resultado em x8. Como não se inicializou nenhum valor em x9, seu valor é nulo, e o resultado se resume a zerar novamente o registrador x8.

Após o estado B, a máquina troca para o C, da classe reg-to-reg, aonde realiza a operação desejada entre os sinais "operando_a" e "operando_b" (operando da ALU), que são os registradores x8 e x9, lidos do banco. O resultado da operação, "alu_out", é armazenado no registrador temporário da ALU, que no estágio seguinte (estado D), é escrito no banco de registradores de acordo com a permissão de escrita do mesmo, "regfile_w".

Durante a execução de ambas as instruções destaca-se o estado do sinal "regacess" durante a leitura, e escrita do banco, estados B e D, comum a ambas as classes. As duas instruções executadas tem permissão de acesso diferente no banco, c.addi visualiza

Figura 4.4 – Execução da instrução c.and da classe reg-to-reg do núcleo multicíclico. Todos os sinais estão representados em hexadecimal, com exceção dos registradores x8 e x9 do banco, sinais "register_file[8]" e "register_file[9]", respectivamente..



Fonte: Autor.

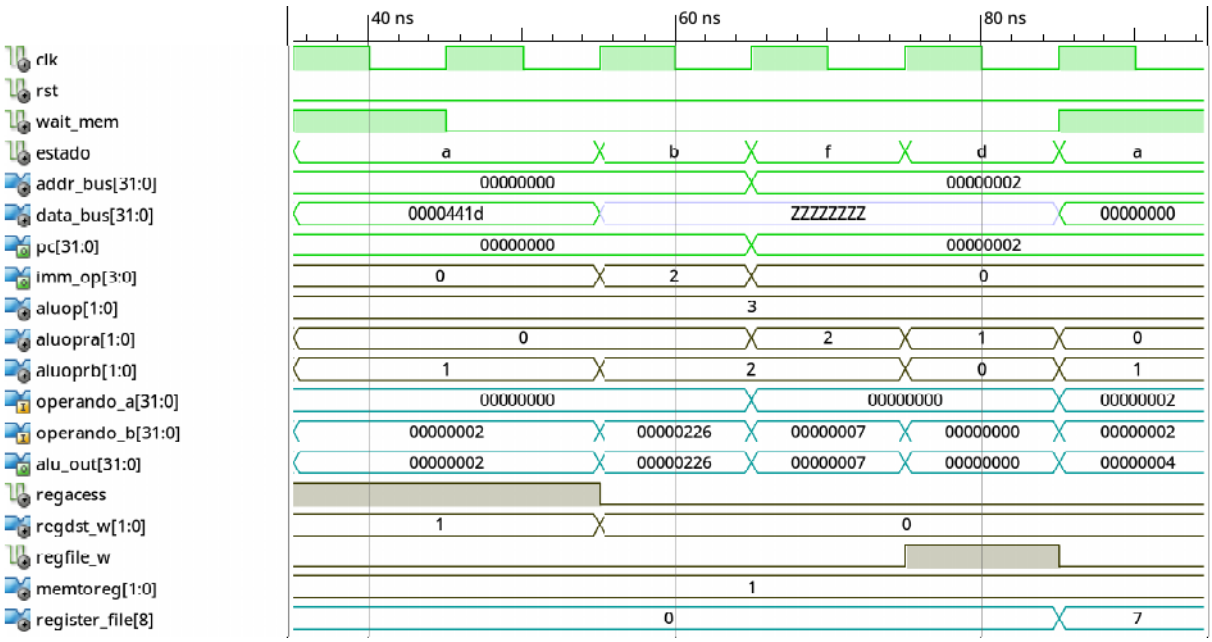
todos os 16 registradores para uso, já c.and é limitado de x8 a x15 do banco. Durante a execução da instrução c.and é perceptível então o nível lógico 1 (ativo) do sinal "regacess", diferentemente do obtido durante a execução de c.addi.

4.2.2.2 Classe LI e MV

Classe contendo as instruções c.li e c.mv (load immediate e move), suas características de execução não possibilitavam o enquadramento em ambas reg-to-reg e reg-to-imm, visto que o operando A da ALU é a constate 0, multiplexada pelo sinal de controle "alu_opra", e o operando B variado para as duas. Na simulação da Figura 4.5 se executou a instrução c.li, a fim de inicializar o registrador x8 com o valor 7.

Após a busca da instrução no estado A da máquina, decodificação e leitura do banco no terceiro ciclo (estado B), se realiza a operação de soma na ALU, entre a constante 0 (operando_A) e o sinal imediato (operando_B). Caso a instrução fosse c.mv, o "operando_b" seria o dado lido no banco de registradores. A soma não alterando em nada o valor do "operando_b" acaba por apenas o transferir para a saída da ALU, "alu_out", que é então armazenada no banco de registradores no estágio seguinte, estado D.

Figura 4.5 – Execução da instrução c.li da classe LI e MV do núcleo multicíclico. Todos os sinais estão representados em hexadecimal, com exceção do registrador x8 do banco, sinal "register_file[8]".



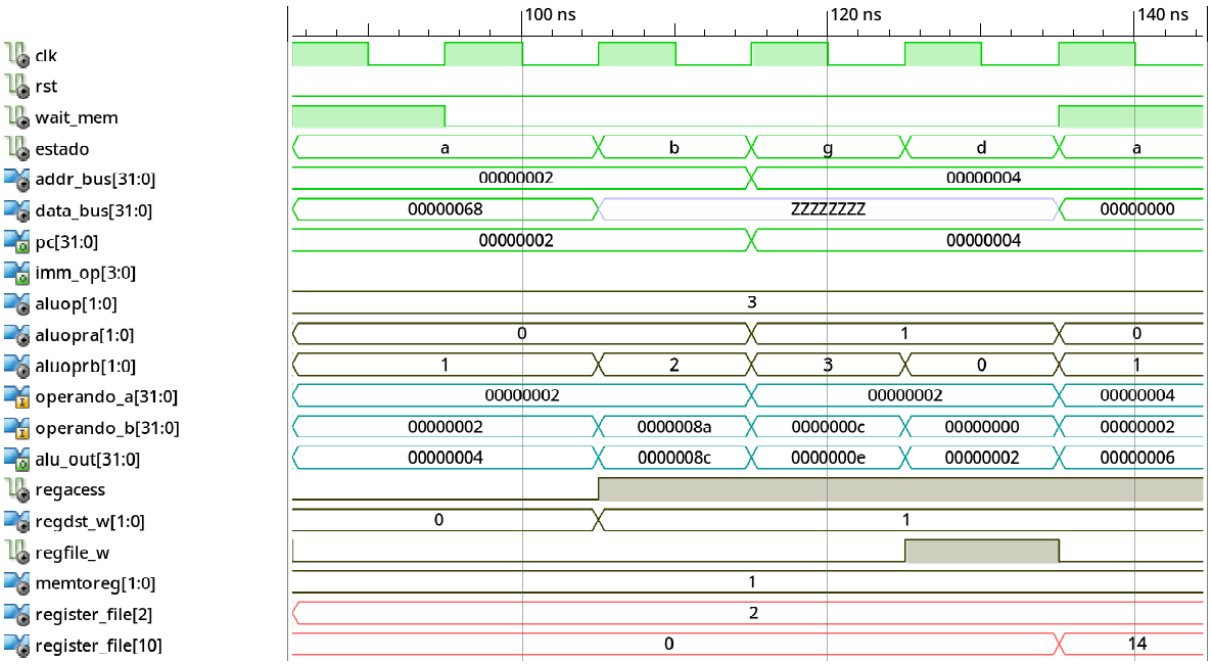
Fonte: Autor.

4.2.2.3 Classe ADDI4SPN

Classe única para a instrução c.addi4spn, mais precisamente essa instrução forma uma operação reg-to-imm, porém devido a diferenças nos sinais de controle (e consequente estados da máquina de mealy), sua operação foi considerada separadamente. A Figura 4.6 contém a simulação da instrução, aonde a mesma soma o registrador x2, considerado o stack pointer pela ABI do RISC-V, com o sinal imediato de valor 12, armazenando o resultado no registrador x10. O registrador x2 já estava previamente inicializado com o valor decimal 2, por meio da instrução c.li.

Pelo digrama de tempo visualiza-se a sequência de estados executada pela unidade de controle no decorrer da instrução, passando pelo estado G após o A e B (comuns a todas as instruções), sendo o estado G utilizado por todas as instruções de acesso a memória. Sua operação sempre é de um registrador do banco no "operando_a" da ALU, e do imediato no "operando_b". Sendo então o estado reaproveitado para o calculo desejado da instrução c.addi4spn, pois a mesma sempre utiliza o registrador x2 como "operando_a", assim como as instruções LWSP e SWSP. No ciclo seguinte, a máquina troca para o estado D, comum a diversas instruções para realização da escrita no banco de registradores, conforme indica o sinal em estado ativo, "regfile_w".

Figura 4.6 – Execução da instrução c.addi4spn do núcleo multicíclico. Todos os sinais estão representados em hexadecimal, com exceção dos registradores x2 e x10 do banco, sinais "register_file[2]" e "register_file[10]", respectivamente.



Fonte: Autor.

4.2.2.4 Classe de Desvios Condicionais

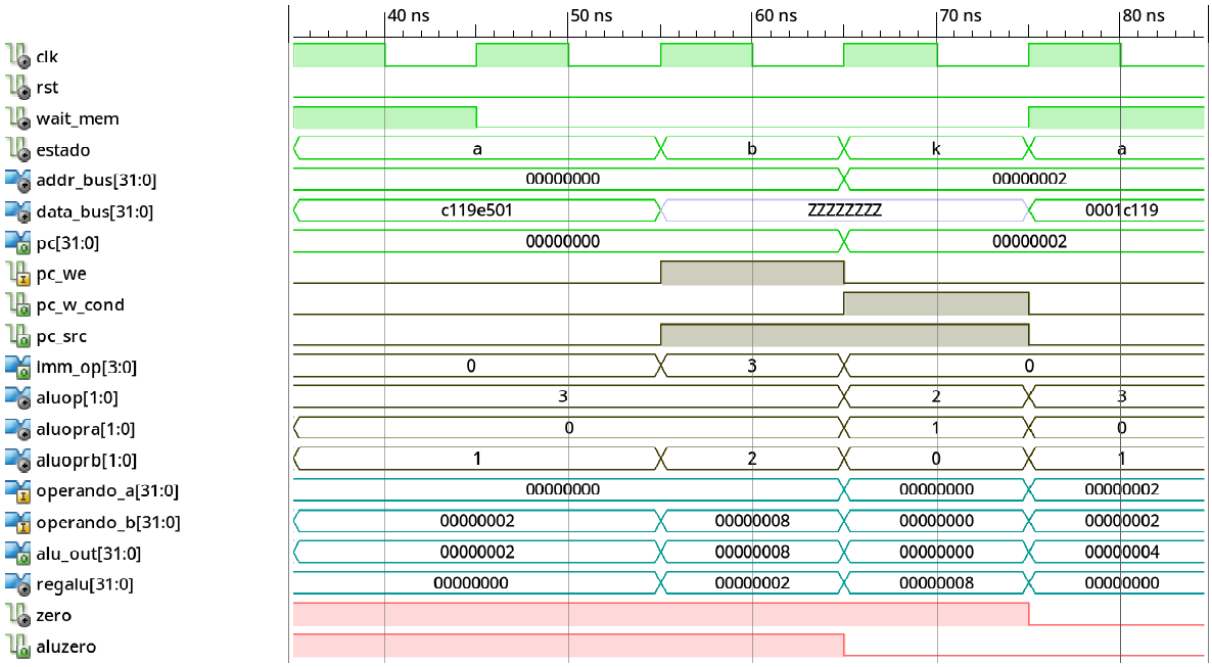
A classe de desvio condicionais é composta pelas duas instruções de branch da extensão C do RISC-V, c.bnez e c.beqz. Na simulação da Figura 4.7 se demonstra a execução da primeira instrução, c.bnez, da seguinte sequência de instruções:

```
PC:

   0: e501                 c.bnez x10,8 <label>
   2: c119                 c.beqz x10,8 <label>
   4: 0001                 c.addi x0,0 # NOP
   6: 0001                 c.addi x0,0

00000008 <label>:
   8: 0001                 c.addi x0,0
```

Como nenhum registrador foi inicializado por meio da instrução load immediate, por exemplo, o desvio de c.bnez (branch if not equal to zero) não ocorre. O mesmo tem como condição que o dado lido do registrador, utilizado como "operando_a" da ALU, seja diferente de 0, no ciclo em que o estado atual da máquina de mealy se define como K. Nesse momento é ativado o sinal de controle "pc_w_cond" para esperar a confirmação da ALU

se a condição é verdadeira ou não, e através de uma OR desse sinal com o "aluzero", se define o resultado da instrução, no caso, negativo (pc_we = 0). O sinal de habilitação de escrita no PC ficou ativo apenas no estado B, para que se atualizasse o seu valor pelo PC incrementado (PC+2), calculado no estado A, e armazenado no registrador temporário localizado na saída da ALU, "regalu".

Figura 4.7 – Execução da instrução c.bnez da classe desvio condicional, do núcleo multiciclico. Todos os sinais estão representados em hexadecimal.

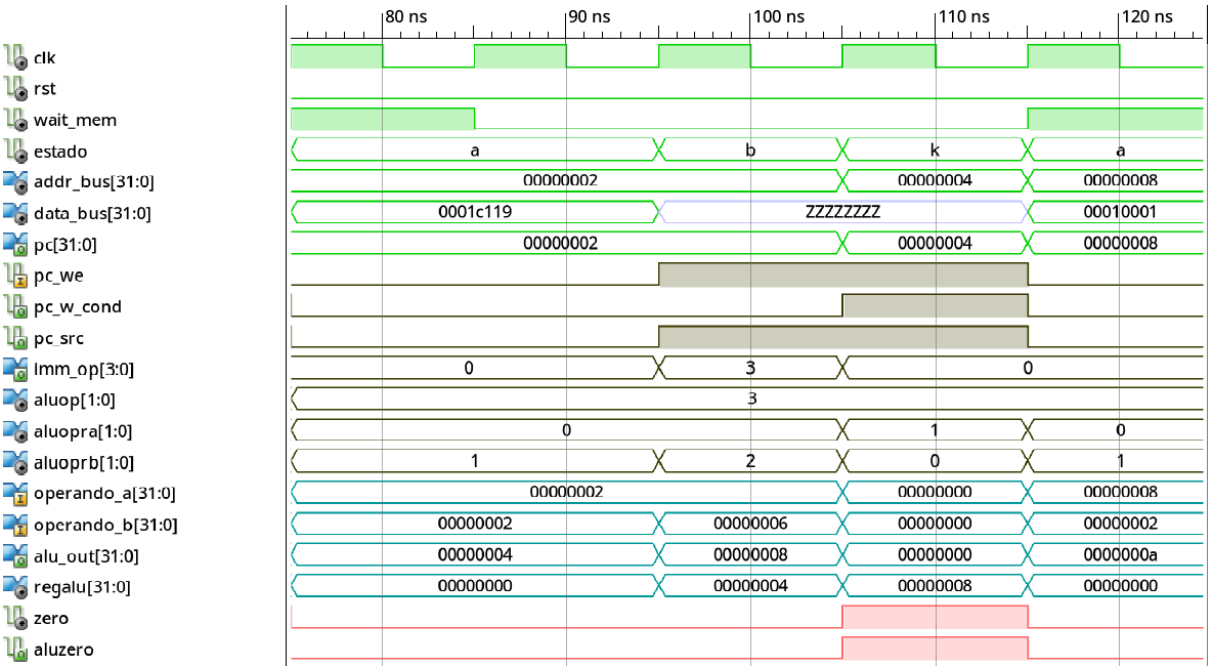


Fonte: Autor.

No estado K também se ativou o sinal "pc_src", de forma a selecionar o endereço alvo de desvio para a entrada de PC, no caso do desvio ocorrer, sendo o endereço alvo calculado durante o estado B da máquina, e armazenado no registrador temporário da ALU, para uso no ciclo seguinte (estado K). O sinal "zero" é o resultado da comparação a 0 do "operando_a" da ALU, porém como a instrução deve desviar apenas se não for igual, o mesmo foi complementado, e renomeado para "aluzero".

O diagrama de tempo da Figura 4.8 apresenta a simulação da instrução c.beqz, que ocorre logo após a c.bnez demonstrada, dessa vez, ocorrendo o desvio. No estado K, após a comparação a 0 do registrador x10, novamente se tem em nível 1 o sinal "zero", porém dessa vez como se trata de uma condição de igualdade, o sinal não é complementado, assim "aluzero"fica ativo, e por consequência a habilitação de escrita do PC também. No estado A da próxima instrução é visível o valor de PC desviado, 8 hexadecimal.

Figura 4.8 – Execução da instrução c.beqz da classe desvio condicional, do núcleo multiciclico. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

4.2.2.5 Classe de Desvio Incondicional PC e REG

Classe de instruções de desvio incondicionais PC, em que se calcula o endereço alvo somando-se o valor atual de PC, com o imediato da instrução, dividindo-se em duas instruções possíveis, c.j ou c.jal. O diagrama de tempo da Figura 4.9 apresenta a execução da instrução c.jal presente no código:

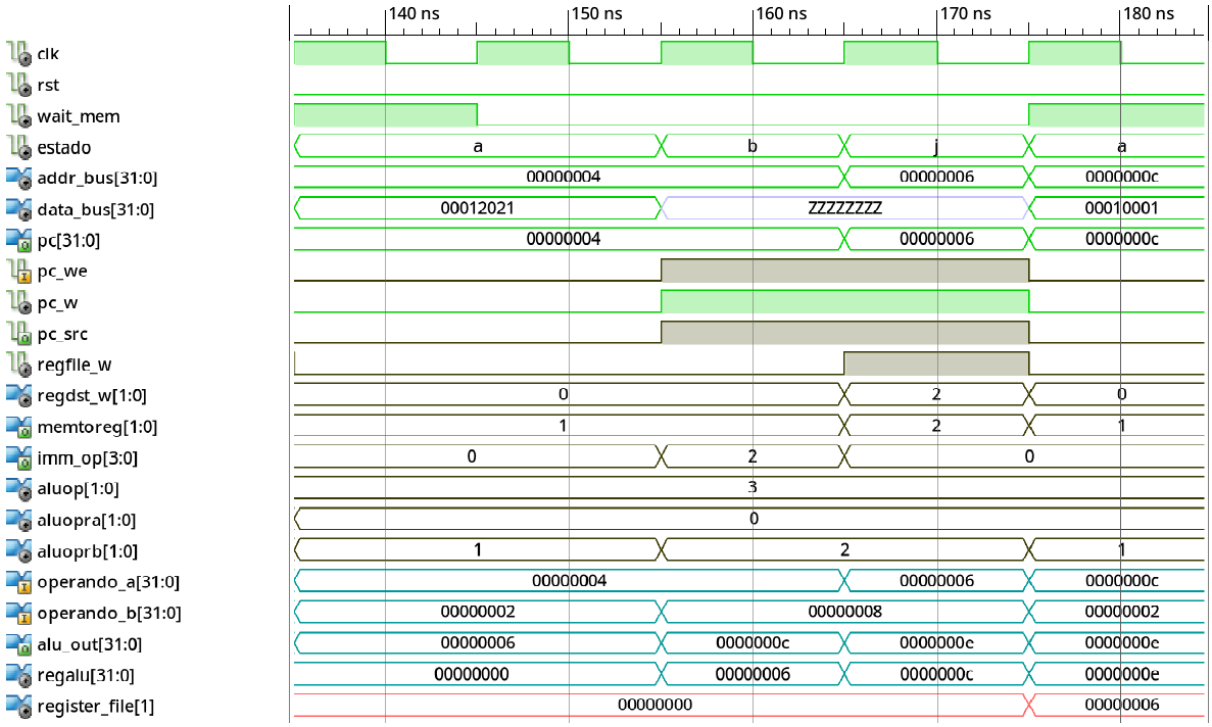
```
  PC:

   0: 0001                  c.addi x0,0 # NOP
   2: 0001                  c.addi x0,0
   4: 2021                  c.jal c <label>
   6: 0001                  c.addi x0,0
   8: 0001                  c.addi x0,0
   a: 0001                  c.addi x0,0

0000000c <label>:
   c: 0001                  c.addi x0,0
```

aonde o valor de PC desvia para o endereço hexadecimal C, enquanto se armazena no registrador x1, nomeado de link register pela ABI do RISC-V, o endereço da instrução que

seria executada após a de desvio, valor 6 hexadecimal. O endereço alvo de desvio é calculado no estado B da instrução, armazenando-se o mesmo no registrador de saída da ALU, "regalu", para posterior uso. No estado seguinte (J) é realizado então o desvio habilitando-se a escrita de PC (pc_we) pelo sinal "pc_w", a seleção do endereço alvo como entrada de dado do PC é realizada pelo sinal de controle "pc_src". Também se realiza em paralelo a escrita no banco de registradores do endereço em sequência ao do jump, no caso, 6, sendo que o registrador destino é fixado no x1, representado pelo sinal "register_file[1]".

Ao término do ciclo J, visualiza-se no estado A da próxima instrução o valor do PC sendo o desejado, assim como a realização do armazenamento de PC no link register, que agora pode ser utilizado como endereço de retorno de rotinas. Caso a instrução fosse uma c.j, a diferença estaria simplesmente no fato do sinal de habilitação de escrita do banco estar desabilitado, "regfile_w=0", ignorando-se assim o dado na sua porta de escrita. O dado de entrada do mesmo é controlado pelo sinal "memtoreg", que nessa classe seleciona a saída de PC como entrada da porta de escrita. Como o registrador destino é uma constante (x1) não informada nos campos da instrução, se utiliza o sinal "regdst_w" para multiplexar o endereço 1 de escrita no banco (rd - registrador destino).

Figura 4.9 – Execução da instrução c.jal da classe desvio incondicional PC, do núcleo multicíclico. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

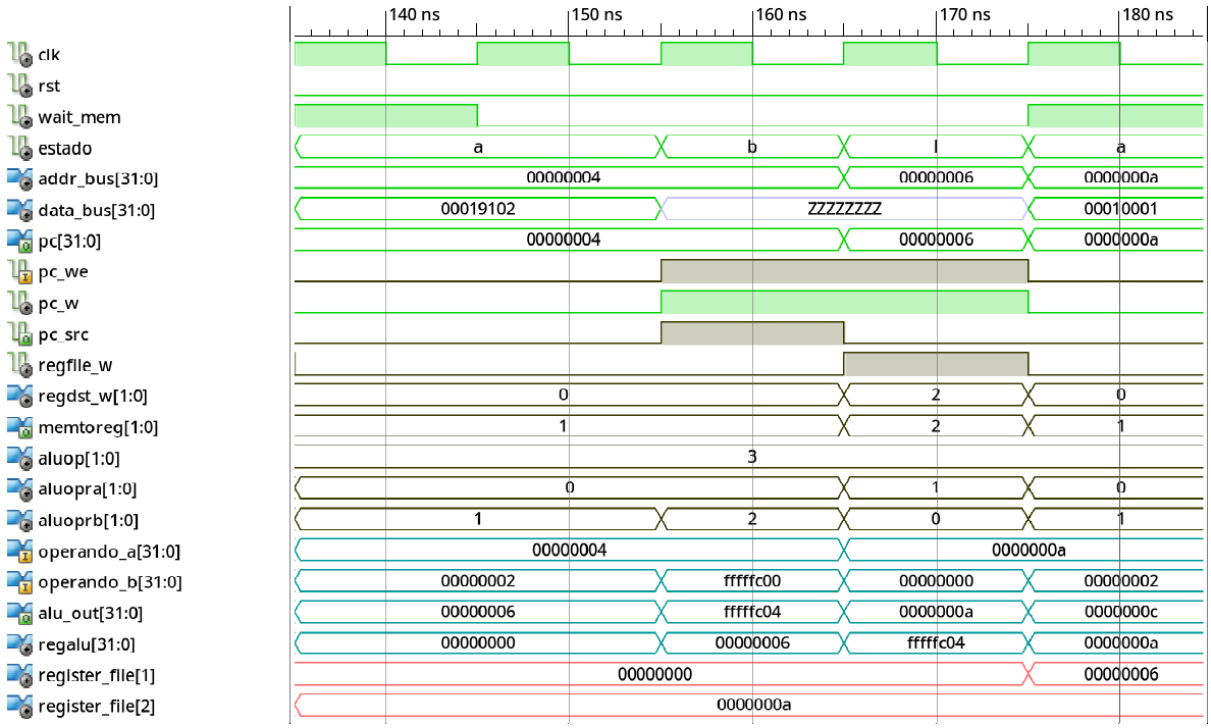
As instruções de desvio incond. REG se comportam da mesma forma, porém o

endereço alvo é o valor contido no registrador lido do banco. O diagrama de tempo da Figura 4.10 possui o resultado da simulação da instrução c.jalr do seguinte programa:

```
   PC:

0: 0001                 c.addi x0,0 # NOP
2: 4129                 c.li x2,10
4: 9102                 c.jalr x2
6: 0001                 c.addi x0,0
8: 0001                 c.addi x0,0
a: 0001                 c.addi x0,0
```

Na simulação, no estado L se realiza o desvio para o valor contido na saída da ALU, "alu_out", que é o resultado da soma do registrador x2, aonde se tem armazenado o endereço alvo (carregado por c.li) com o registrador x0 (constante 0). A constante 0 da soma (x0) esta definida de forma fixa na codificação da instrução. No ciclo seguinte, pode ser verificado o valor desejado no PC, assim como no link register, pois c.jalr também armazena endereço da próxima instrução apenas no registrador x1.

Figura 4.10 – Execução da instrução c.jal da classe desvio incondicional REG, do núcleo multicíclico. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

4.2.2.6 Classe de Leitura da Memória

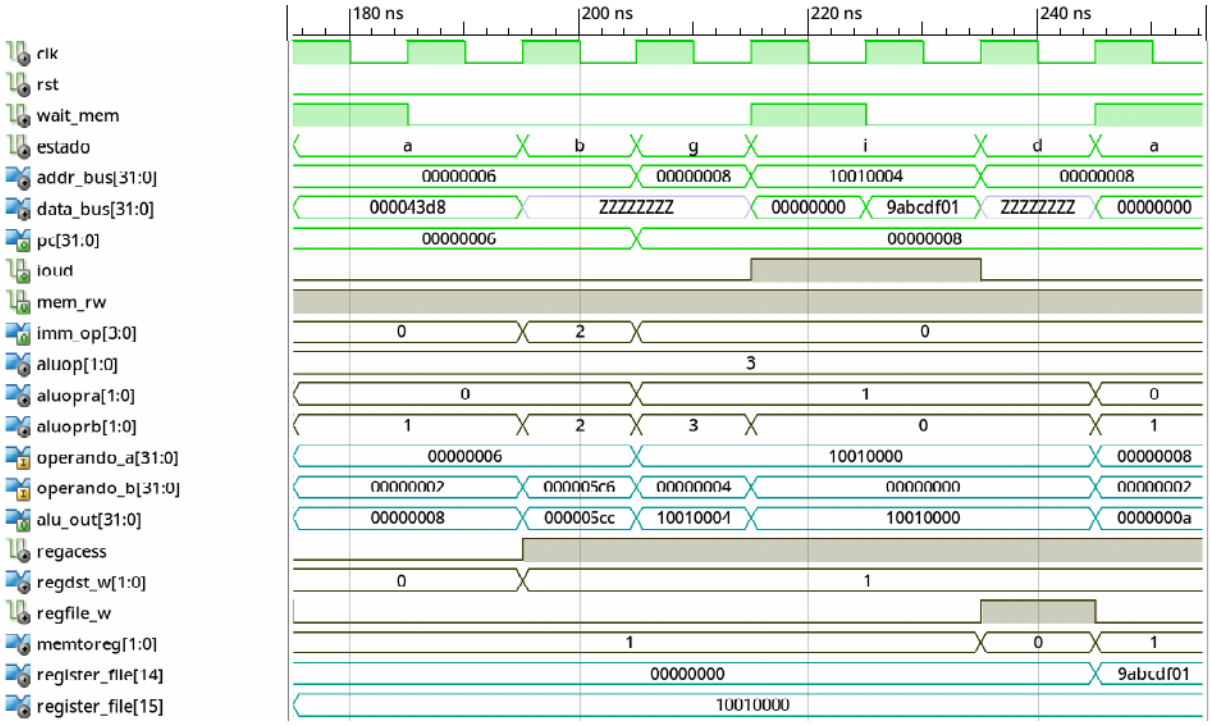
A classe de leitura da memória possui as instruções c.lw e c.lwsp, demonstrando-se na Figura 4.11 a simulação da instrução c.lw. O código executado foi:

```
 addr:

section .text

0: 6785                 c.lui x15,0x1 # 3 primeiras instr. carregam end. da seção .
2: 0785                 c.addi x15,1
4: 07c2                 c.slli x15,0x10
6: 43d8                 c.lw x14,4(x15)

section .data

 10010000 <_data1>:
 10010000: 5678
 10010002: 1234

 10010004 <_data2>:
 10010004: df01
 10010006: 9abc
```

Quando a máquina de estados entra no estado G, é calculado o endereço de acesso a memória, com base no "operando_a" (dado lido do banco de regs.) e o "operando_b" (sinal imediato). O dado que se deseja buscar é o indexado por "_data2", armazenado no endereço 0x1001004, sendo visualizado o mesmo no resultado da ALU, "alu_out". No próximo ciclo, estado I, ocorre o acesso a memória, que envia o sinal "wait_instr", a fim de travar o núcleo enquanto o dado não esta disponível no barramento. Para selecionar a saída do registrador temporário da ALU (que contém o resultado do ciclo anterior), é colocado em nível lógico 1 o sinal "ioud". No próximo ciclo de clock se tem então, no barramento "data_bus" o dado desejado, que é então armazenado no registrador temporário de dados do núcleo. Durante todo o acesso o sinal "mem_rw" permanece em nível 1, pois do contrário, iria indicar uma ação de escrita para a memória.

O ciclo final da instrução de load realiza a escrita do dado no banco de registradores, estado D, aonde percebe-se que o sinal "regacess" está ativo, limitando o acesso de x8 a x15 (assim como também estava na leitura, estado B). O dado do registrador temporário de dados da memória é multiplexado para a porta de escrita do banco pelo sinal de controle "memtoreg=00", e a escrita então habilitada (regfile_w) e realizada na próxima borda de subida do clock. É possível conferir no registrador x14 (register_file[14]) do banco o dado

Figura 4.11 – Execução da instrução c.lw da classe leitura da memória, do núcleo multicíclico. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

buscado da memória.

4.2.2.7 Classe de Escrita da Memória

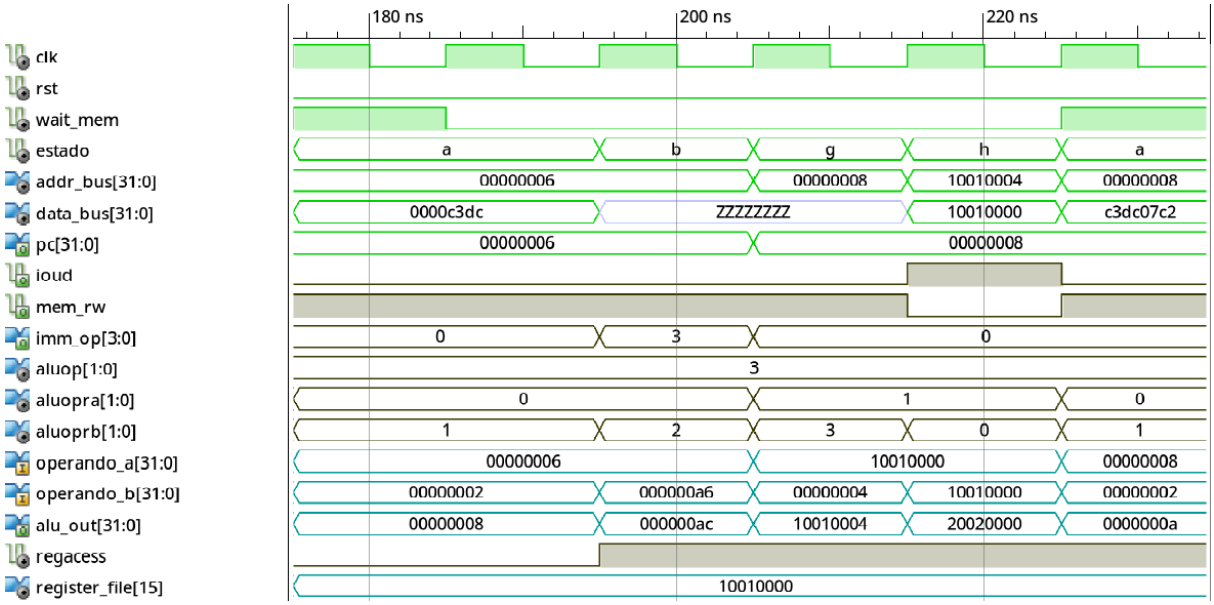
Classe de instruções de escrita, contendo as seguintes possibilidades: c.sw e c.swsp. A simulação da Figura 4.12 demonstra o diagrama de tempo da execução da instrução c.sw, do programa:

```
addr:

0: 6785                 c.lui x15,0x1 # 3 primeiras instr. carregam end. da seção .
2: 0785                 c.addi x15,1
4: 07c2                 c.slli x15,0x10
6: c3dc                 c.sw x15,4(x15)
```

O estado A e B são utilizados para busca e decodificação/leitura do banco, respectivamente, sendo então no estado G determinado o endereço alvo de acesso. Para calcular o endereço alvo se utilizou de um dado lido do banco, como endereço base, e do sinal imediato da instrução, sendo visualizado o resultado na saída da ALU, "alu_out". Armazena-se

então o endereço alvo no registrador de saída da ALU, que no próximo ciclo (estado H) é selecionado como endereço do barramento no lugar da saída do PC, por meio do nível 1 no sinal "ioud". A escrita da memória é habilitada por meio do nível lógico baixo no sinal "mem_rw".

Figura 4.12 – Execução da instrução c.sw da classe escrita da memória, do núcleo multicíclico. Todos os sinais estão representados em hexadecimal.



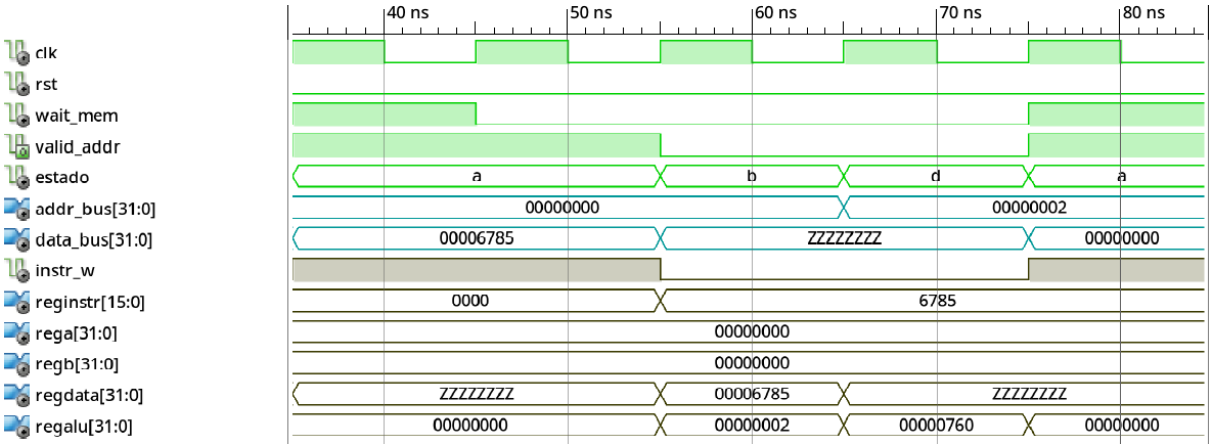
Fonte: Autor.

4.2.2.8 Dependência da Memória

A simulação da Figura 4.13 contém o comportamento do núcleo multicíclico diante da latência de um ciclo para acesso a memória (RAM e ROM), para uma instrução qualquer em execução. No primeiro ciclo do estado A é realizado o acesso da memória, que então devolve em nível ativo o sinal "wait_data", travando por um ciclo a escrita em todos os registradores temporários do núcleo. Os sinais "rega" e "regb" representam os registradores de saída A e B do banco, "regdata" o registrador temporário para leitura de dados da memória, "regalu" o registrador de saída da ALU, e por fim "reginstr" o registrador de armazenamento da instrução a ser executada.

Assim como os registradores temporários não escrevem nenhum dado no primeiro ciclo do estágio A, a máquina de estados também é travada, mantendo estável o acesso por mais um ciclo. No segundo ciclo então o sinal "wait_data" é colocado em nível em 0, com o dado da instrução disponível então nos 16 LSB do barramento de dados, "data_bus".

Figura 4.13 – Simulação demonstrando o funcionamento da espera do núcleo pelo dado da memória.



Fonte: Autor.

A instrução é armazenada então em "reginstr" na próxima borda de subida do clock com a habilitação da mesma (sinal "instr_w=1"), sendo visualizada no estado B.

Para evitar que o núcleo faça acessos constantes a memória no decorrer da execução de cada instrução, que ocorreria a cada 2 ciclos (um de latência, outro de leitura), se enviar o sinal "valid_addr=0" para o decodificador da memória. Essa ação desabilita então os acessos, colocando em nível de alta impedância o barramento de dados. Dessa forma, os únicos estados em que o sinal o endereço no barramento é valido (valid_addr=1), são os estados A para leitura de instrução, I e H para leitura e escrita de dados, respectivamente.

4.2.3 Núcleo Pipeline - RV32EC

Verifica-se na Tabela 4.3 o consumo de recursos da FPGA, para ambas as implementações do sistema completo, e do núcleo unicamente, observando-se que se atingiu a frequência minima desejada de 100 MHz. O resultado completo de utilização dos recursos do dispositivo FPGA Spartan-6 estão disponíveis nos Apêndices B.3 e B.4, e a descrição em VHDL do hardware no Apêndice F.

Na hierarquia de sua escrita em VHDL, diagrama da Figura 4.14, se visualiza apenas os componentes internos do núcleo, separados pelas suas posições ao longo dos estágios do pipeline, com exceção da unidade de forward que atua ao longo de diversos estágios realizando a realimentação de dados. Para o preditor de desvios localizado no estágio IF, se descreveu dois blocos de RAM, um contendo apenas o endereço alvo, e outro o restantes das informações (tag, bit de estado e valor do PC).

Tabela 4.3 – Resumo de utilização de recursos da FPGA Spartan-6 para a versão do caminho de dados com pipeline.

Recursos	Disponível	Usado	Utilização (%)
Núcleo			
Número de Slice Registers	18.224	975	05%
Número de Slices LUTs	9.112	1.660	18%
Número de Slices Ocupados	2.278	605	26%
Blocos de RAM	32	2	6%
Frequência Máxima (MHz)		100,371	
Sistema Completo			
Número de Slice Registers	18.224	1.137	06%
Número de Slices LUTs	9.112	2.137	23%
Número de Slices Ocupados	2.278	782	34%
Blocos de RAM	32	18	56%
Frequência Máxima (MHz)		100,09	

Fonte: Autor.

A execução do caminho de dados pode ser visualizada ao dividi-la em diferentes classes de instruções, aonde a instrução AUIPC em específico se caracterizou separadamente, conforme a Tabela 4.4.

Tabela 4.4 – Classes de instruções do pipeline.

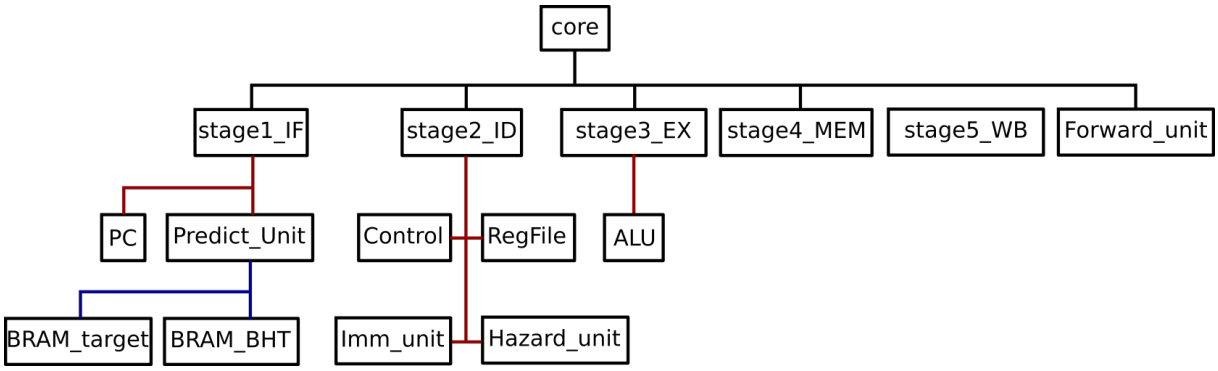
Classe	Característica
Reg-to-Reg	operações entre dois registradores do banco
Reg-to-Imm	operações entre um registrador do banco e o imediato
AUIPC	instrução add upper immediate to PC
Desvio incond.	instruções jump
Desvio cond.	instrução branch
Load	leitura da memória
Store	escrita da memória

Fonte: Autor.

4.2.3.1 Classe Reg-to-Reg e Reg-to-Imm

Na execução da classe de instruções que realizam a operação entre dois registradores do banco, reg-to-reg, se tem as seguinte possibilidades de instruções: ADD, SUB, OR, AND, XOR, SLL, SRL, SRA e SLT e SLTU. A Figura 4.15 demonstra a execução de ambas as classes (reg-to-reg e reg-to-imm). No estágio um, se tem o incremento de PC conforme indicado pelo sinal "rvc", e então sua escrita na próxima borda de subida do sinal de relógio, assim como também a escrita da instrução lida da memória no registrador IF/ID.

Figura 4.14 – Hierarquia da descrição em VHDL do núcleo com a implementação da técnica de pipeline. Se utilizou dois blocos de RAM para uso do preditor de desvios condicionais, um somente para armazenamento do endereço alvo, utilizando o outro para armazenar as informações restantes.



Fonte: Autor.

No segundo estágio então se realiza a leitura do banco de registradores, e decodificação da instrução em paralelo, a unidade imediato é ignorada nesse caso.

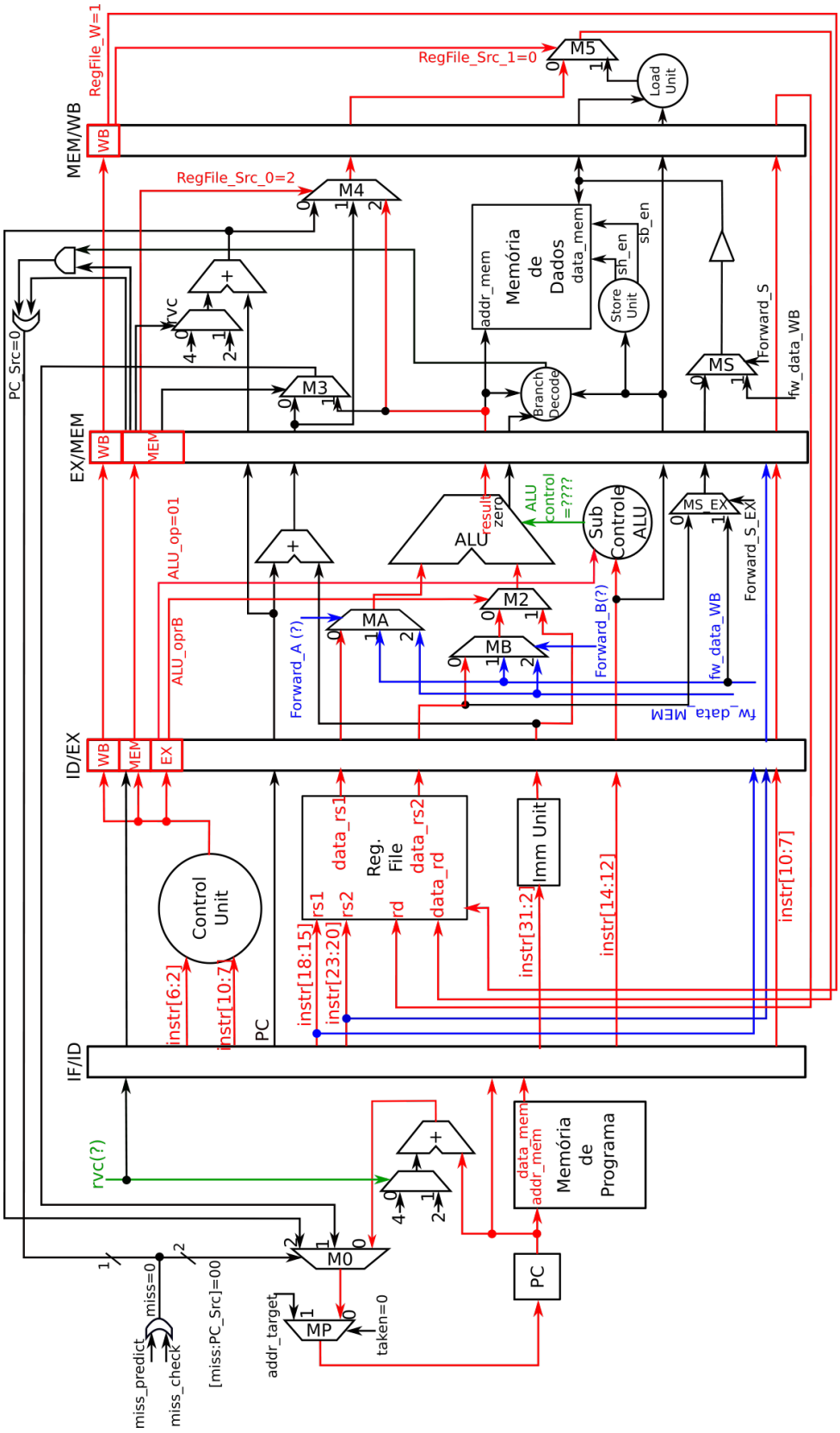
Informações relevantes da instrução são passadas ao estágio seguinte, sendo elas os endereços de acesso ao banco (rs1, rs2 rd) para uso da unidade de forward, assim como o campo [12:10] da instrução para decodificar a operação da ALU na sua subunidade dedicada. No estágio EX o valor de "ALU control" é indefinido, visto que depende de qual instrução da classe em específico se está executando, diferenciado pelo valor do campo funct3 ([12:10]), indicado pelo valor constante do sinal de controle "ALU_op=01". Um dos operando da ALU também é definido de forma fixa, sendo a saída do banco de registradores, "ALU_oprB=0". No caso da necessidade de forward, deixou-se os sinais "Forward_A" e "Forward_B" em estado indefinido. O quarto estágio consiste simplesmente na seleção do valor a ser enviado ao quinto estágio de escrita, que corresponde a saída da ALU através do MUX 4, sendo controlado pelo sinal "RegFile_Src_0=2".

E por fim no último estágio é necessário selecionar o mesmo dado novamente, e não o lido da memória, logo se obtém "RegFile_Src_1=0", e habilita-se a escrita pelo sinal "RegFile_W=1". Em suma, os valores dos sinais relevantes a classe estão de acordo com a Tabela 4.5. Outros sinais não especificados, ou não interferem no pipeline (logo seu valor não importa), ou estão desabilitados (caso de store, load e desvios).

O diagrama de tempo da Figura 4.16 demonstra a execução de uma instrução de adição reg-to-reg, aonde os dois primeiros ciclos formam a leitura da instrução (1 ciclo de latência da memória), seguidos pelos estágios ID, EX, MEM e WB. Como o pipeline executa diversas instruções simultaneamente, outras são finalizadas no intervalo demonstrado, sendo o código ASM executado composto pelas instruções:

```
PC:
```

Figura 4.15 – Execução de instruções da classe Reg-to-Reg e Reg-to-Imm.



Fonte: Autor.

Tabela 4.5 – Sinais de controle na execução da classe reg-to-reg.

Sinal	Valor
ALU_oprB	0
ALU_op	01
RegFile_Src_0	10
RegFile_Src_1	0
RegFile_W	1

Fonte: Autor.

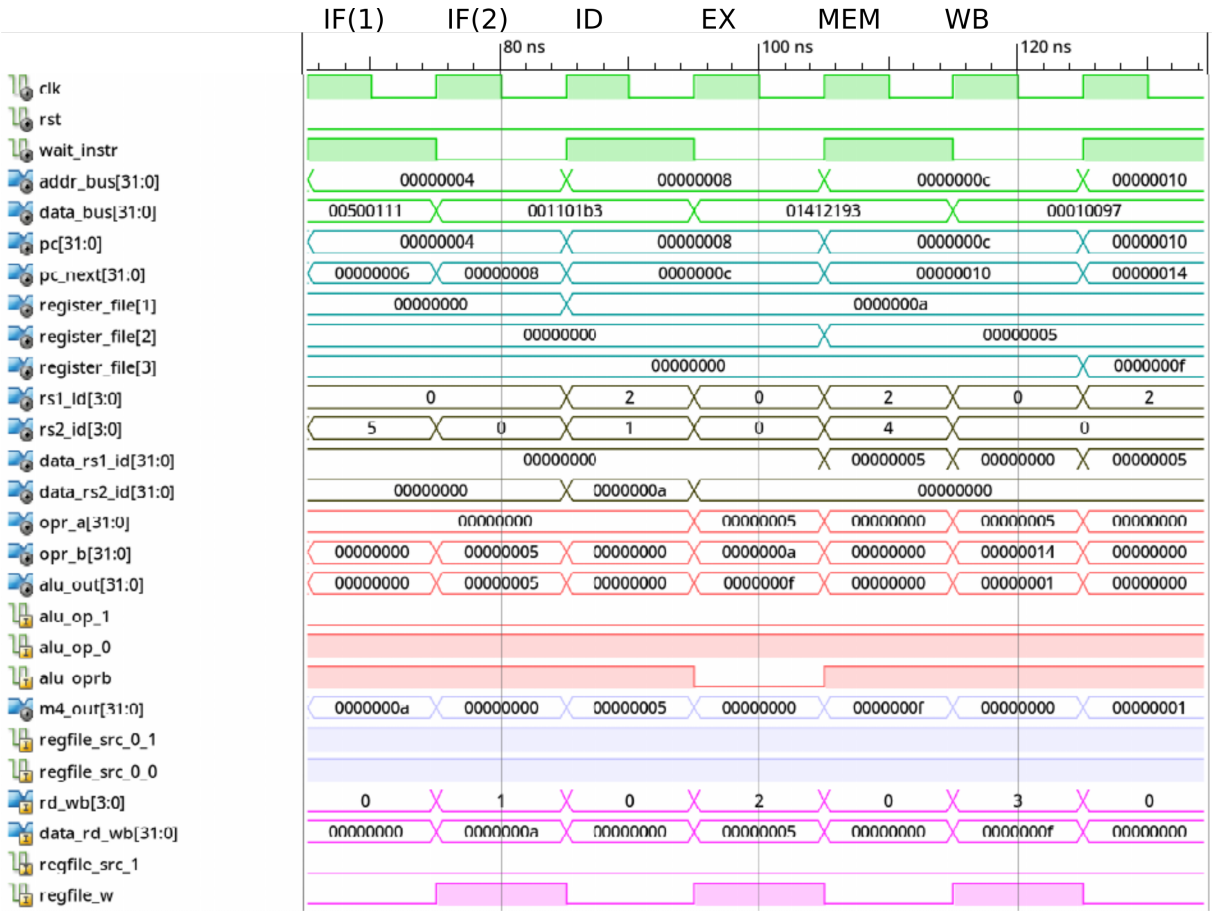
```
0: 40a9                 c.li x1,10
2: 4115                c.li x2,5
4: 001101b3            add x3,x2,x1
8: 01412193            slti x3,x2,20
```

assim, conforme se obtém no diagrama, o resultado da adição é 15, armazenado no registrador (sinal "register_file[3]") x3 do banco. Os sinais de controle essenciais da classe estão em exibição no diagrama, como o de habilitação de escrita do banco, "regfile_w", que aparece em nível 1 três vezes, duas para cada instrução load immediate, e uma última para a adição.

As instruções que executam operações entre um registrador do banco e o sinal imediato (reg-to-imm) se resumem a: ADDI, NOP, ORI, ANDI, XORI, SLLI, SRLI, SRAI, SLTI e SLTIU. A funcionalidade do caminho é a mesma que para a classe reg-to-reg, com exceção do valor do sinal "ALU_oprB" possuir agora o valor 1, selecionando assim sempre o imediato decodificado no estágio dois, nesse caso não se ignora a unidade de imediato. Uma instrução reg-to-imm em especifico utiliza um valor diferente para o sinal de controle "ALU_op", sendo ela a LUI (load upper immediate). A mesma apenas transfere para a saída da ALU o sinal imediato a ser armazenado no banco de registradores, logo, necessita de uma codificação dedicada para indicar essa operação a ALU. O sinal "ALU_op" é colocado então em nível "11", com prioridade sobre o campo [12:10] da instrução, resultando no sinal "ALU control=1010". Como a instrução LUI apenas transfere para a saída o operando imediato, ignora-se a leitura do banco para a execução desta instrução.

Seguindo o mesmo código exemplificado para o diagrama de tempo simulador para instruções reg-to-reg, a quarta instrução é da classe reg-to-imm, e seu resultado (no mesmo programa) é visualizado na simulação da Figura 4.17. A operação armazena no registrador x3 do banco o valor 1, caso se prove verdadeira a condição de que o registrador x2 é menor que o imediato decimal 20 (com sinal). Sendo verdadeiro nesse caso, o resultado obtido é 1.

Figura 4.16 – Simulação de uma instrução da classe reg-to-reg do núcleo pipeline, ADD, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.



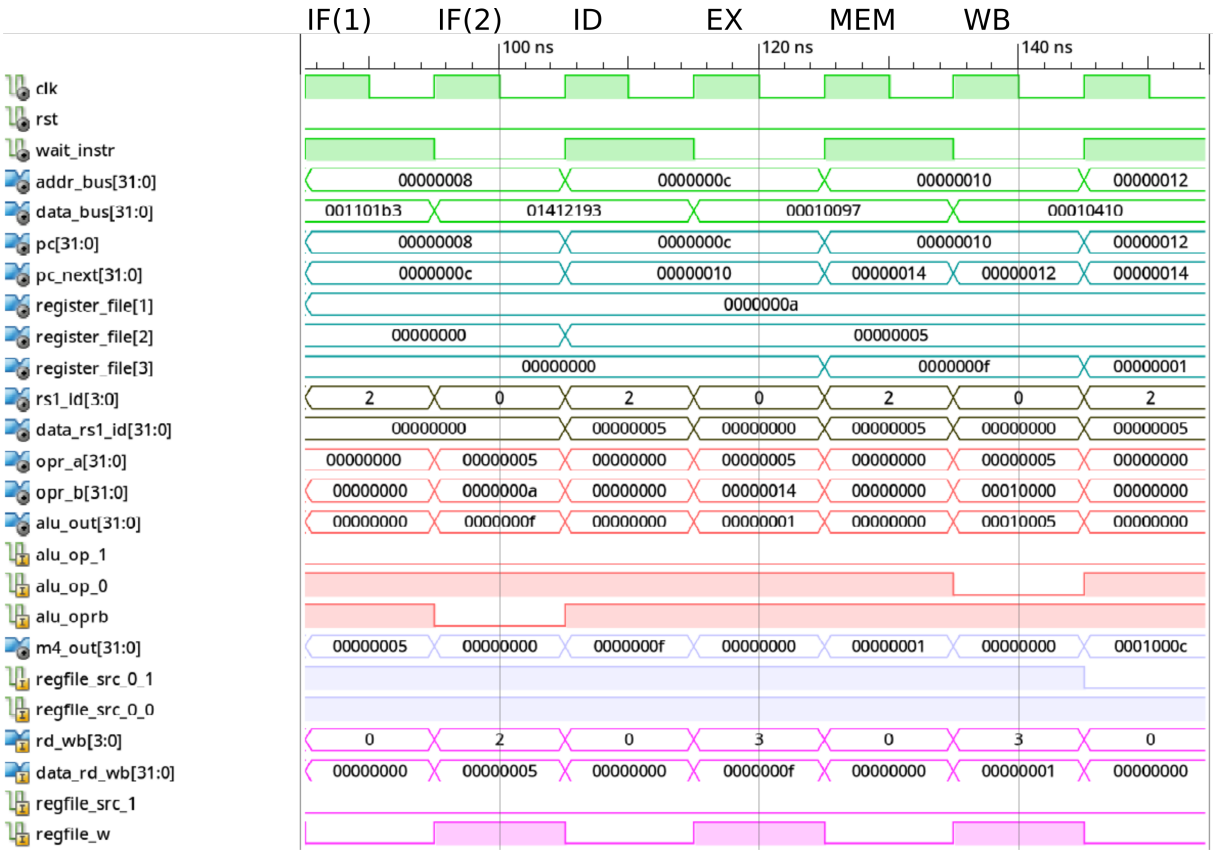
Fonte: Autor.

4.2.3.2 Classe AUIPC

Instrução Add Upper Immediate to PC (AUIPC), que é tecnicamente uma instrução do tipo reg-to-imm, visto que realiza a soma do sinal imediato ao registrador dedicado PC, porém se separou devido a propagação de dados por um caminho diferente do núcleo, como demonstra a Figura 4.18. Nessa instrução, após a leitura da instrução na memória, e incremento do PC, se tem no ciclo seguinte a decodificação da instrução, e do campo imediato em paralelo. Como o registrador é o PC, que se propaga pelo pipeline, a leitura do banco é desnecessária. Uma unidade de soma dedicada é localizada no estágio de execução, a mesma serve para calculo de desvios, somando o PC ao sinal imediato, sendo então reaproveitado para execução desta instrução, a ALU é ignorada nesse ciclo.

Após o término do terceiro estágio (EX), apenas se propaga o resultado do somador dedicado no estágio EX, pela saída do MUX 4, logo o sinal "RegFile_Src_0" possui valor 1. No estágio final (WB), deve-se selecionar novamente o valor propagado para a saída

Figura 4.17 – Simulação de uma instrução da classe reg-to-reg do núcleo pipeline, SLTI, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.



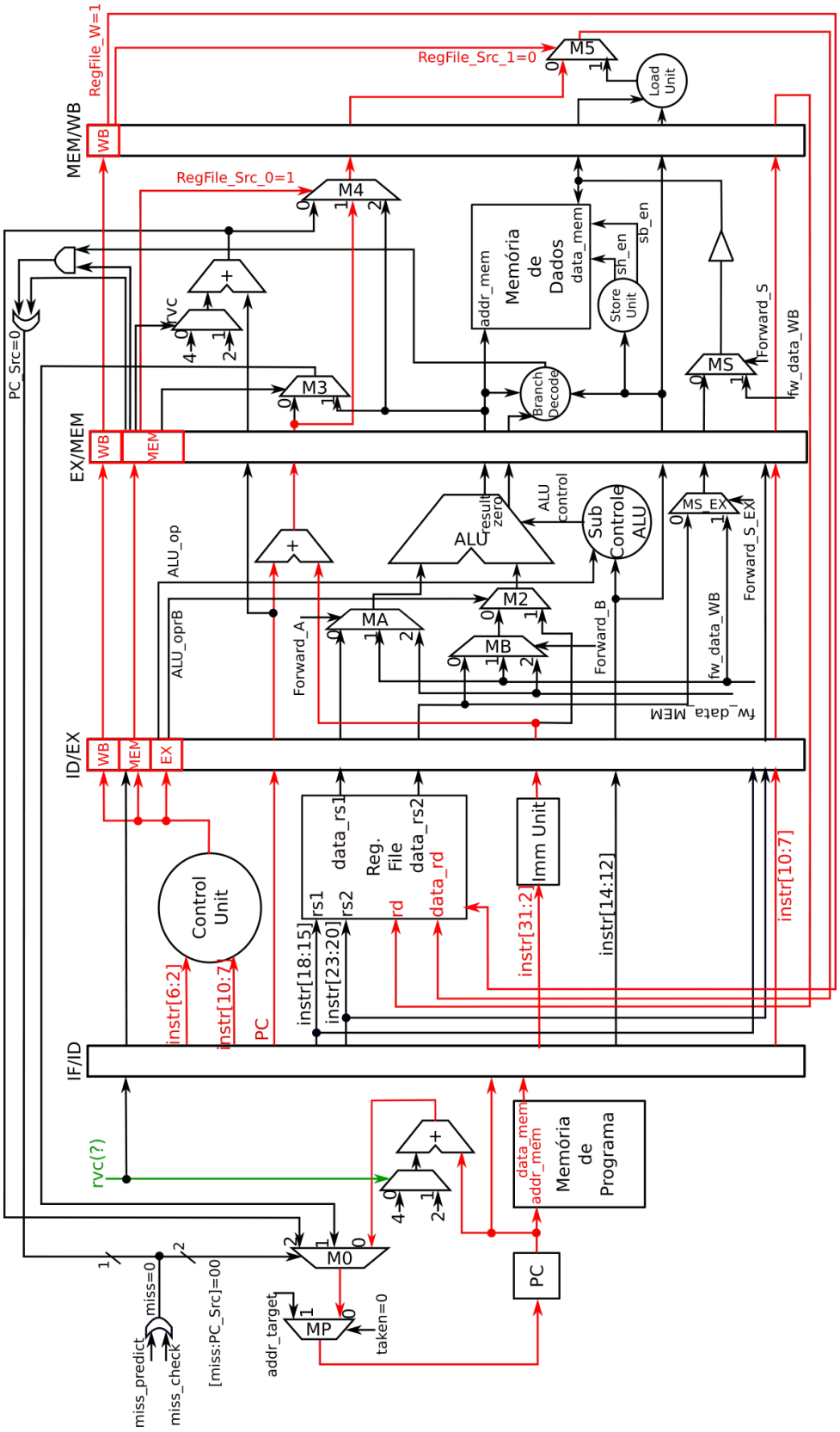
Fonte: Autor.

do MUX 5, sendo assim, obtém-se "RegFile_Src_1=0", e para a escrita do resultado no banco de registradores, "RegFile_W=1". O diagrama de tempo da Figura 4.19 demonstra a execução da instrução AUIPC conforme:

```
PC:

c: 00010097          auipc x1,0x10
```

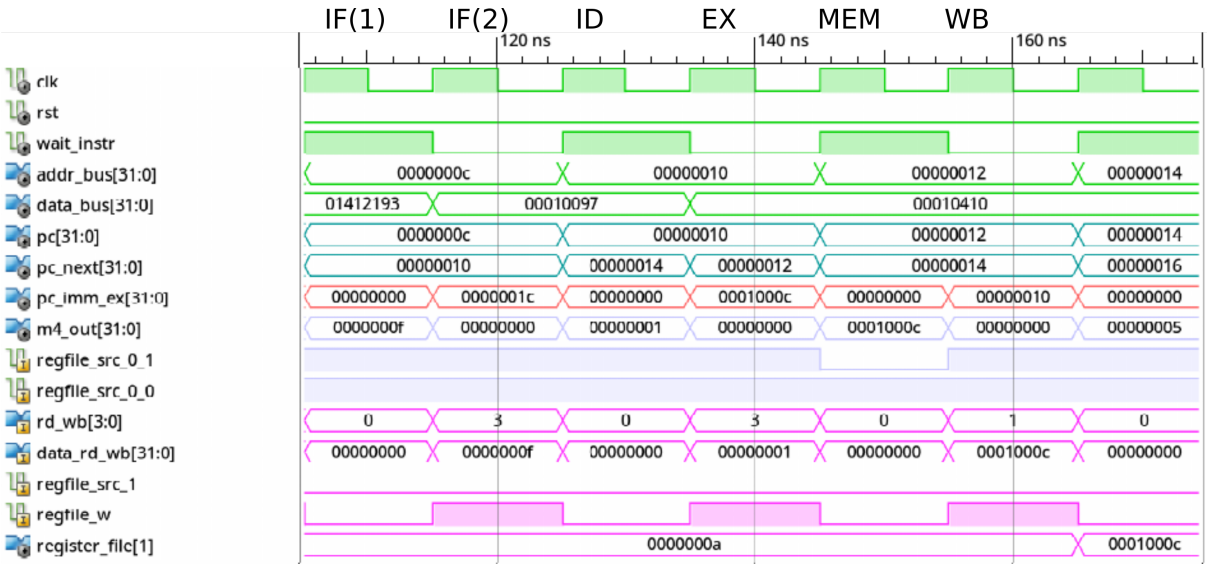
aonde o valor de PC para a instrução (em hexadecimal) é C, que ao somar com o imediato gerado, resulta no valor 0001000C. Ressaltando que o imediato desta instrução é adicionado aos 24 bits superiores de PC.

Figura 4.18 – Execução da instrução add upper immediate to PC (AUIPC).



Fonte: Autor.

Figura 4.19 – Simulação da instrução AUIPC do núcleo pipeline, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.



Fonte: Autor.

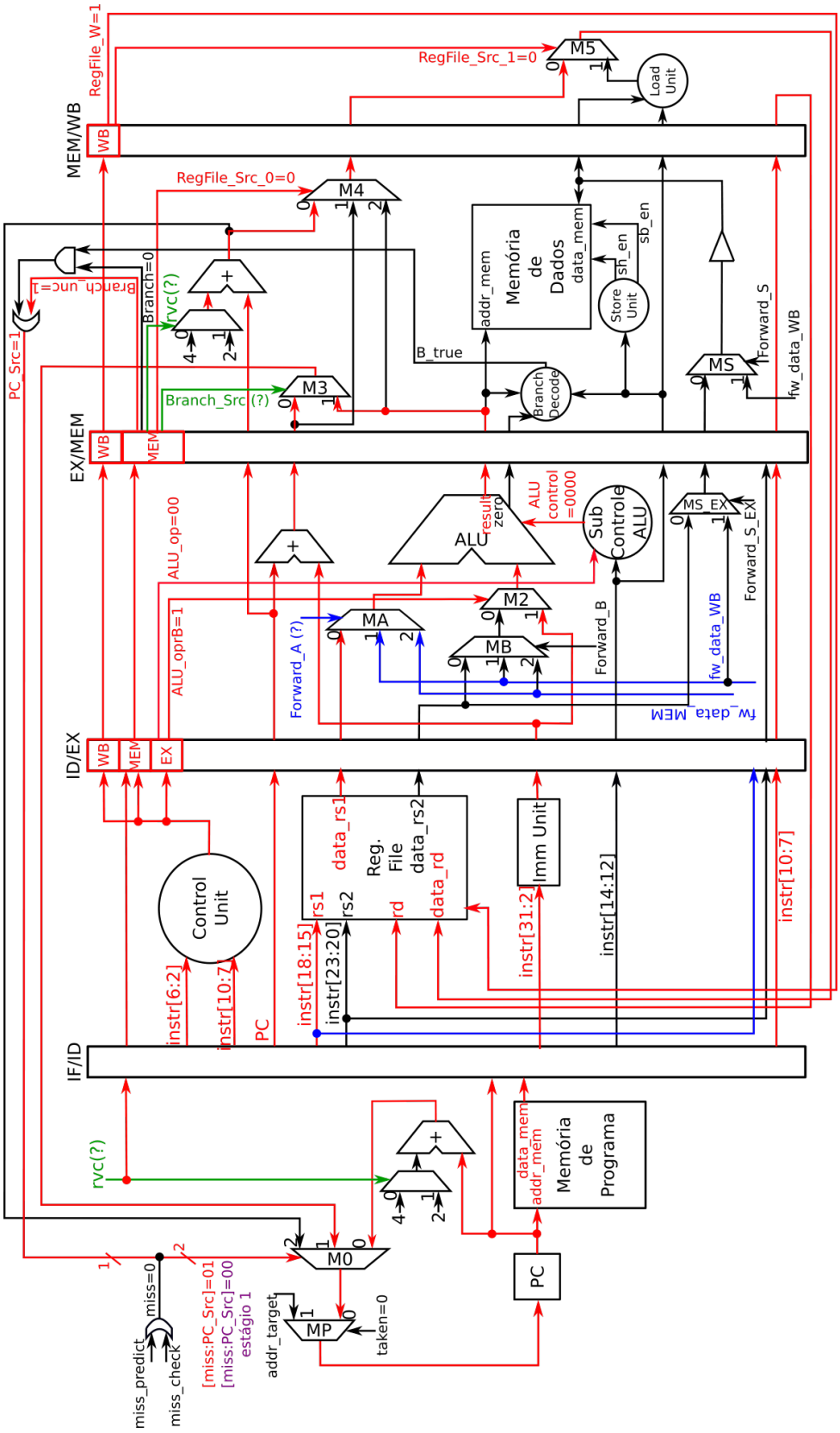
4.2.3.3 Classe de Desvio Incondicional

Instruções JAL e JALR, que realizam um desvio incondicional, armazenando o valor atual do PC no banco de registradores, conforme a execução da Figura 4.20. No primeiro estágio se realiza a leitura da instrução, e incremento do PC, especificou-se o estado do sinal concatenado [miss:PC_Src] em dois estágios diferentes, o primeiro, e quarto. Pois no primeiro o mesmo deve selecionar o incremento do PC para ser escrito no mesmo, e no quarto estágio, o endereço de desvio calculado, sendo os valores, respectivamente 00 e 01. No segundo estágio se realiza a decodificação da instrução e do imediato, sendo a leitura do banco de registradores relevante apenas para a instrução JALR.

No terceiro estágio o caminho das duas instruções diverge, para JAL se utiliza do somador dedicado, calculando-se assim o endereço alvo com o PC propagado pelo pipeline, e o imediato decodificado, aqui os sinais "ALU_op" e "ALU_oprB" não interessam. No quarto estágio então deve ser selecionado o endereço alvo para a saída do MUX 3, que para JAL, corresponde ao sinal de controle "Branch_Src=0". O sinal "PC_Src" é colocado em nível 1 por uma porta OR com o sinal "Branch_unc=1".

Para a instrução JALR o caminho utilizado, partindo novamente do terceiro estágio, é o da ALU. Nela se realiza a soma do campo rs1 lido no banco de registradores, com o sinal imediato, logo os sinais "ALU_oprB" e "ALU_op" possuem os valores 1 e 00, respectivamente. Seleciona-se então a saída da ALU como endereço alvo no quarto estágio, por meio do nível lógica 1 no sinal de controle "Branch_Src", os sinais "PC_Src" e

Figura 4.20 – Execução das instruções de desvio incondicional, JAL e JALR.



Fonte: Autor.

"Branch_unc" comportam-se da mesma forma.

Definido o endereço alvo, ambas as instruções realizam a mesma tarefa final, a de armazenar o valor de PC somado ao seu incremento no banco de registradores. Como o valor propagado de PC não é o de seu incremento, ele deve ser incrementado novamente no estágio MEM, sendo seu valor de acordo com o sinal "rvc"(entre 4 ou 2 decimal). O resultado desse incremento é transmitido para a saída do MUX 4 com o sinal de controle "RegFile_Src_0" em nível 0. No estágio seguinte (WB), "RegFile_Src_1=0" garante que esse dado seja o de entrada para escrita no banco, que é habilitada com "RegFile_W=1".

O diagrama de tempo da Figura 4.21 possui a simulação de uma instrução jump, de acordo com o programa:

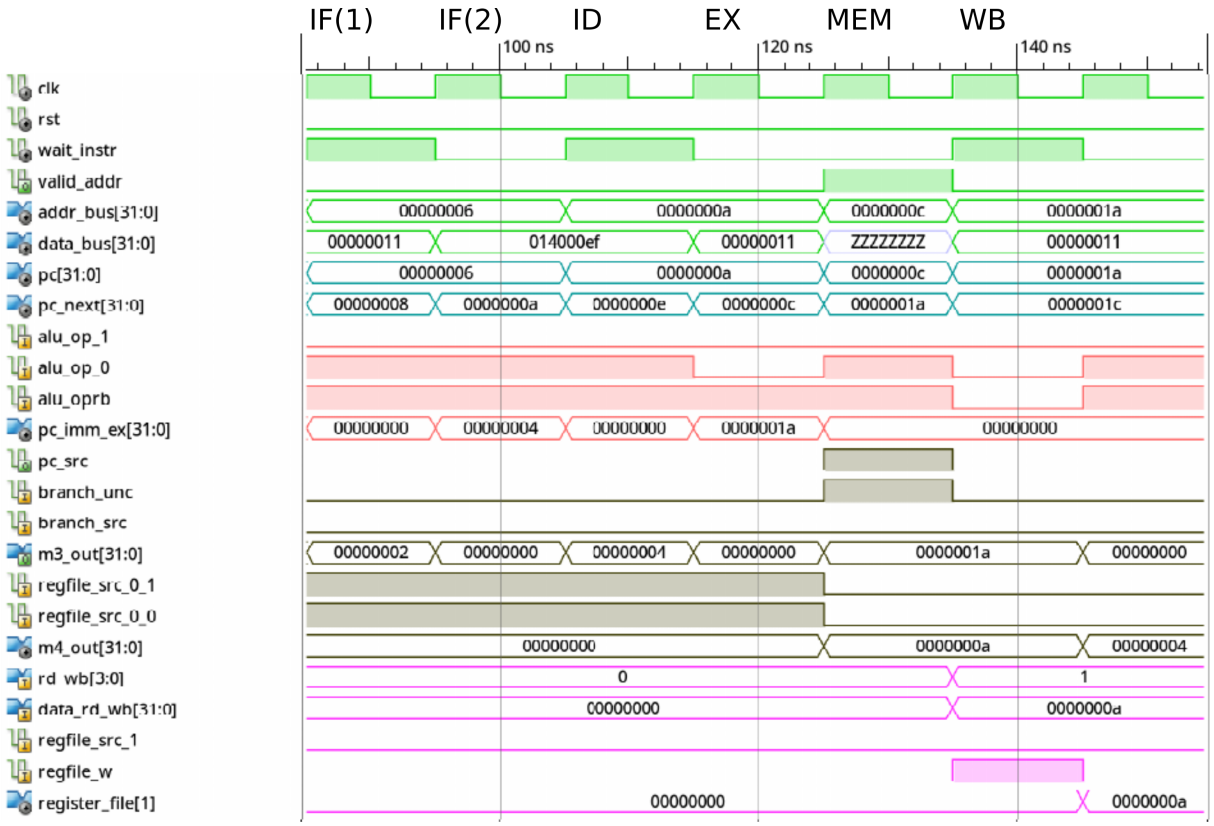
```
  PC:

   0: 0001                  c.addi x0,0 # nop
   2: 0001                  c.addi x0,0
   4: 0001                  c.addi x0,0
   6: 004000ef            jal x1,a <point1>
   a: 0001                  c.addi x0,0

0000001a <point1>:
   1a: 0001                  c.addi x0,0
   1c: 0001                  c.addi x0,0
   1e: 0001                  c.addi x0,0
```

Fica claro no quarto estágio (MEM) a ação dos sinais de controle "pc_src" e "branch_unc" para a seleção do endereço de desvio, e por consequência, também se ativa o sinal "valid_addr"para cancelar a leitura da instrução que seria realizada nesse ciclo no estágio IF. Como é uma instrução JAL, o sinal "branch_src" possui valor 0, multiplexando a saída do somador dedicado a desvios para o estágio IF, e não a saída da ALU. No quinto estágio se ativa a escrita do banco de registradores, a fim de se armazenar o valor de PC da próxima instrução a de desvio, no registrador x1, para que possa ser utilizada como retorno de rotinas. Uma implementação de desvio incondicional em que se ignora o armazenamento de PC seria com registrador destino de escrita x0, pois a ação seria ignorada.

4.2.3.4 Classe de Desvio Condicional

Classe que contém as instruções de desvio condicional: BEQ, BNE, BGE, BGEU, BLT e BLTU. Utiliza um caminho semelhante ao da instrução de desvio incondicional JAL, o diagrama da Figura 4.22 demonstra sua execução no pipeline. No estágio um se tem

Figura 4.21 – Simulação de uma instrução da classe de desvio incondicional do núcleo pipeline, JAL, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.

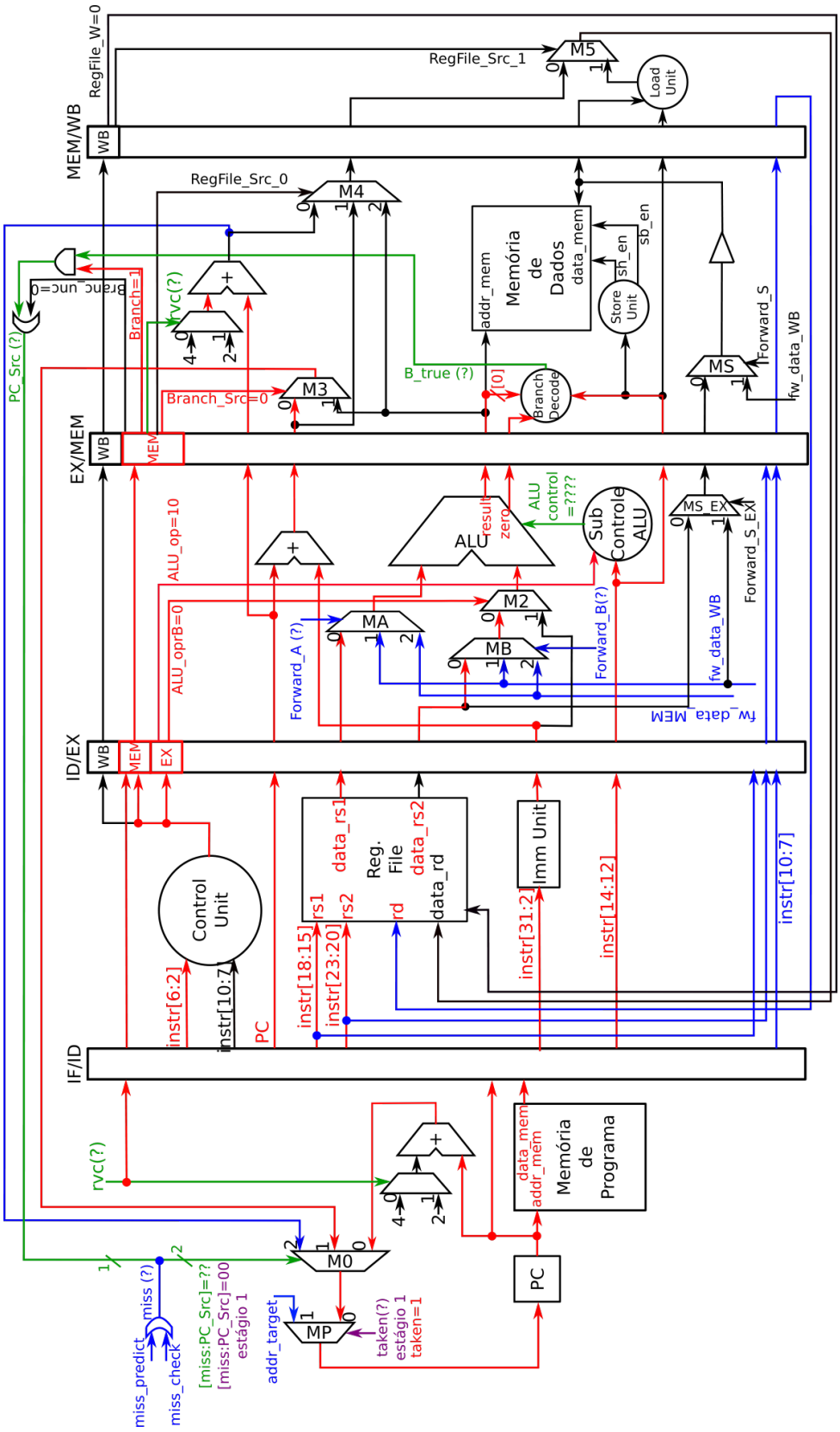


Fonte: Autor.

a busca da instrução, e incremento do PC em paralelo, considerando que nenhum desvio está em ocorrência no estágio MEM do pipeline, a entrada de PC será o seu incremento, de acordo com o valor 00 para o sinal concatenado [miss:PC_Src] no estágio 1. No segundo estágio se utiliza todas a unidades em paralelo disponíveis, decodifica-se a instrução, se realiza a leitura do banco de registradores para comparação entre dois registradores de acordo com o tipo de branch em execução, e também é formado o sinal imediato. O campo [14:12] irá definir o tipo de branch no quarto estágio, dentro da unidade dedicada branch decode, logo seu valor é propagado até o mesmo.

No estágio seguinte, EX, a ALU e o somador dedicado operam em paralelo, sendo a ALU responsável pela comparação dos registradores lidos no banco, a comparação (maior ou igual, menor, igual, diferente) é de acordo com o seu sinal de controle, "ALU control". O o sinal da unidade de decodificação "ALU_op=10", indica prioridade ao campo [14:12] da instrução. O operando da ALU selecionado pelo MUX 2 deve ser o lido do banco, assim se obtém "ALU_oprB=0". O somador dedicado para cálculos de desvio apenas soma o imediato gerado com o valor do PC que é propagado pelo pipeline.

Figura 4.22 – Execução das instruções de desvio condicional.



Fonte: Autor.

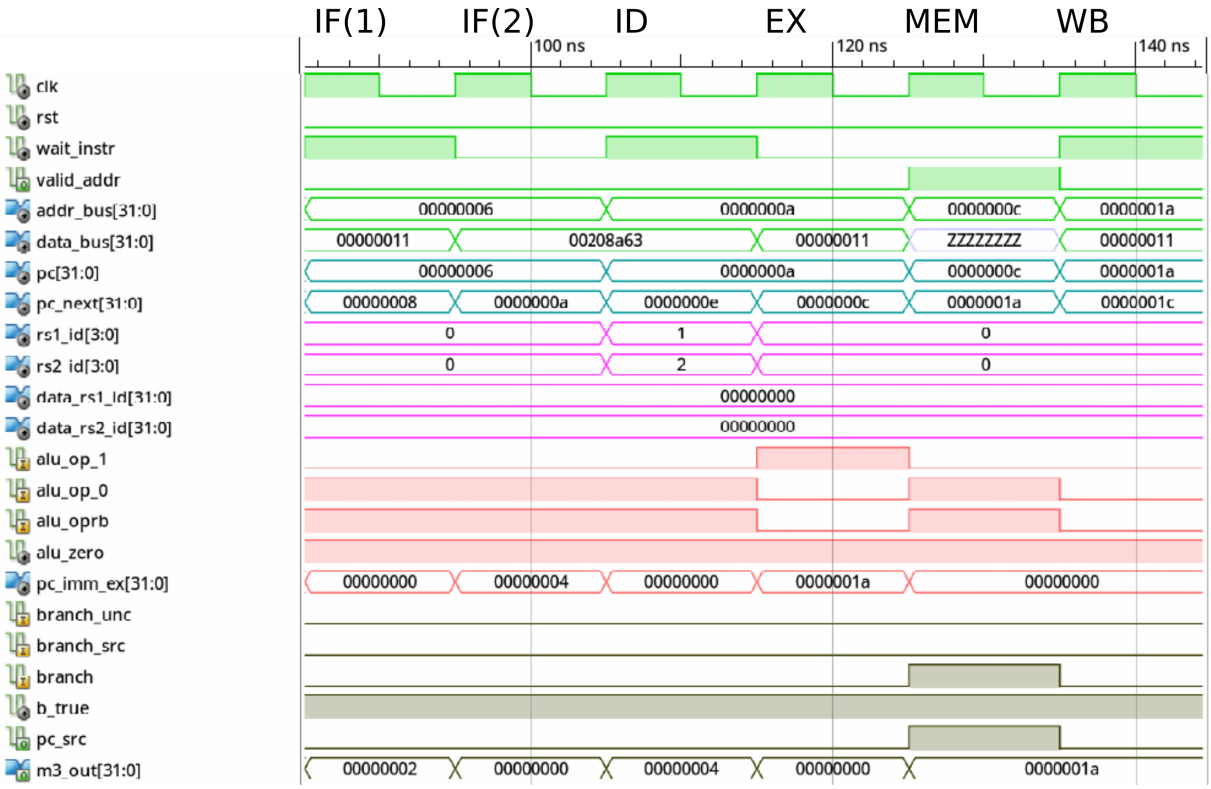
Por fim, no quarto estágio (MEM), o sinal de controle "Branch_Src=0" transmite para a saída do MUX 3 o endereço calculado no somador dedicado. Para que ocorra o desvio, o sinal "PC_Src" deve estar em nível 1, por meio de uma operação AND entre "Branch_=1", e o resultado da comparação, indicado pelo sinal "B_true". A fim de evitar desvios desnecessários, "Branch_unc" está em nível lógico 0. O resultado da comparação é baseado no sinal "zero" da ALU do estágio EX, o LSB de seu resultado (aproveita-se a operação de instruções SLT[U]) e novamente o campo [14:12] da instrução.

Diferente de instruções de desvio incondicional, os branch são armazenados na tabela do preditor dinâmico (de acordo com seu funcionamento), assim sendo, o valor do PC da próxima instrução deve ser enviado ao desvio em execução é enviado ao MUX 0 do estágio IF. Isso é utilizado para correção do endereço alvo, caso o predito dinâmico tenha errado sua ação, selecionando então o mesmo, ao colocar o sinal miss em nível 1. O valor corrigido deve ser incrementado de acordo com o sinal "rvc" no estágio MEM, pois o valor propagado pelo pipeline é o do endereço do branch, e não a instrução seguinte. Além do PC incrementado novamente, o valor do PC também é enviado novamente para o estágio IF, para ser armazenada sua tag na BHT do preditor, quando necessário. O quinto estágio é ignorado nessa classe, e não possuí nenhum efeito, para isso se desabilita a permissão de escrita no banco, "RegFile_W=0'.

O diagrama de tempo da Figura 4.23 apresenta a execução de uma instrução BEQ, que compara dois registradores não inicializados, x1 e x2. Sendo assim, a condição é verdadeira pois ambos são iguais a 0, e o desvio ocorre de acordo com o nível lógico dos sinais "b_true" que é gerado com o sinal "alu_zero". Destaca-se também a importância do sinal "branch", pois sem o mesmo, ocorreria desvios indesejados em momentos em que não se tem instruções branch, como se percebe pelo fato do sinal "b_true" estar ativo em todos os ciclos do intervalo demonstrado. Isso ocorre devido aos valores nulos iniciais do banco, o código executado foi:

```
  PC:

   0: 0001                  c.addi x0,0 # nop
   2: 0001                  c.addi x0,0
   4: 0001                  c.addi x0,0
   6: 00208a63             beq x1,x2,1a <point1>
   a: 0001                  c.addi x0,0

0000001a <point1>:
  1a: 0001                  c.addi x0,0
  1c: 0001                  c.addi x0,0
  1e: 0001                  c.addi x0,0
```

Figura 4.23 – Simulação de uma instrução da classe de desvio condicional do núcleo pipeline, BEQ, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.



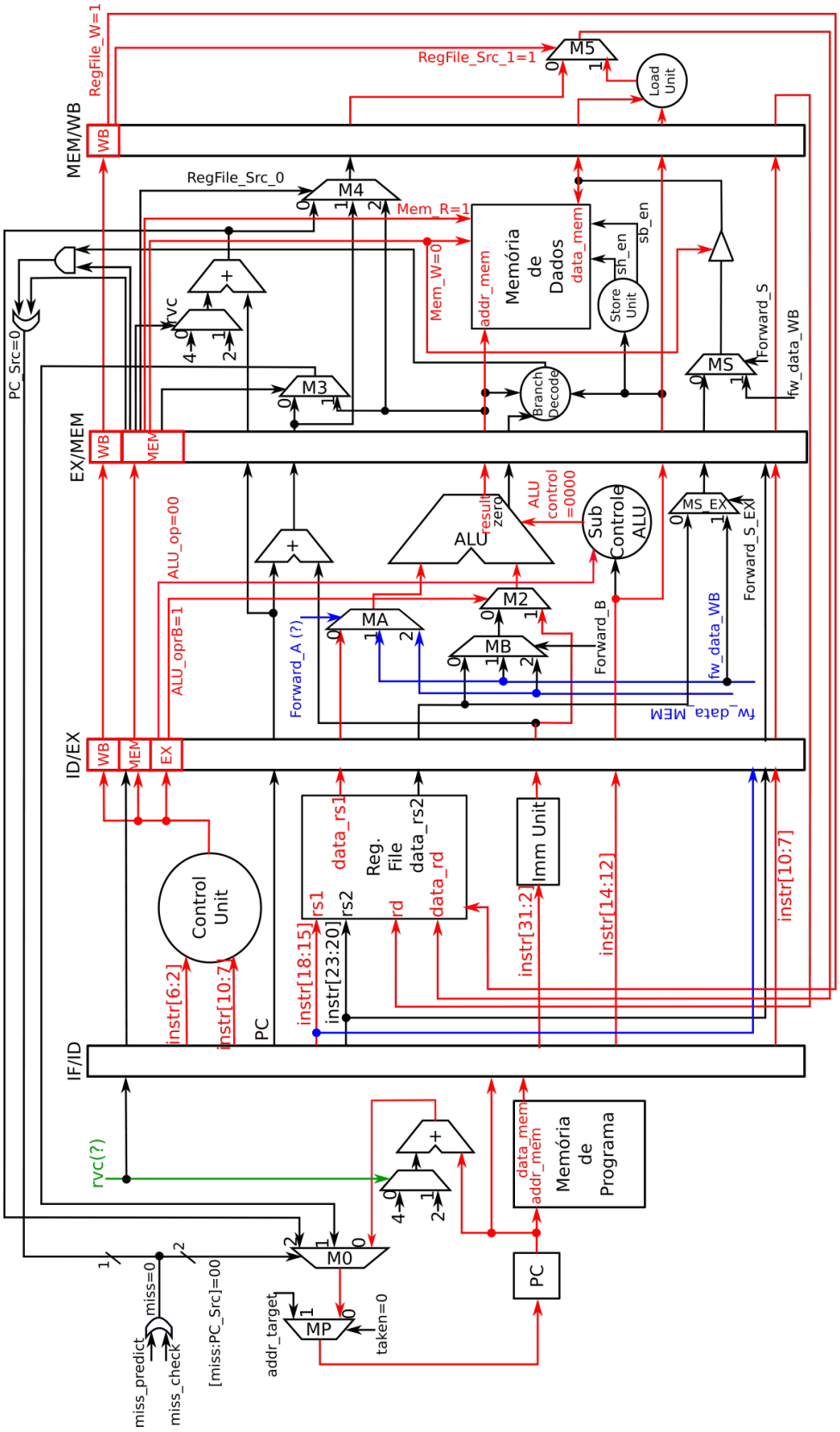
Fonte: Autor.

4.2.3.5 Classe de Load

Classe que contém as instruções de leitura da memória: LW, LH, LHU, LB e LBU. A Figura 4.24 apresenta a classe em execução no caminho de dados. No estágio um se realiza a leitura da instrução da memória, e incremento do PC, que é transmitido a sua porta de entrada pelos multiplexadores 0 e P (considerando que nenhum desvio ocorra adiante no pipeline). O segundo estágio realiza as operações de decodificação da instrução, formação do sinal imediato, e leitura do banco em paralelo, sendo que no banco apenas o endereço dado pela porta "rs1" é relevante, pois é utilizado como endereço base de acesso a memória. O campo [14:12] é propagado pelo pipeline até o quinto estágio, aonde diferencia os tipos de load na sua unidade dedicada.

No terceiro estágio (EX) se realiza a soma do registrador lido no banco, com o sinal imediato, formando assim o endereço alvo de acesso da memória, para indicar a operação de adição o sinal "ALU_op" possui codificação 00, que é dedicada a operação de soma e possui prioridade sobre o campo [14:12] na subunidade de controle da ALU. O operando selecionado pelo MUX 2 deve ser o sinal imediato, logo se obtém "ALU_oprB=1". A leitura

Figura 4.24 – Execução das instruções de acesso a memória para leitura.



Fonte: Autor.

ocorre então no quarto estágio do pipeline, com o endereço alvo calculado, indicando-se a operação pelo sinal de controle "Mem_R=1", e para evitar conflitos é desabilitada a escrita da memória, "Mem_W=0". Por fim, no quinto estágio se utiliza da subunidade de load para extensão do dado buscado da memória quando necessário, sendo a extensão pelo MSB ou 0, indicado pelo campo [14:12] propagado ao longo do pipeline. A escrita do dado no banco é habilitada pelo sinal "RegFile_W=1", e o dado multiplexado para sua porta de entrada de dados pelo sinal de controle "RegFile_Src_1=1".

Na Figura 4.25 se visualiza o diagrama de tempo da simulação de uma instrução LB, em que se executou o seguinte código, com uma variável inicializada na memória de dados (para leitura):

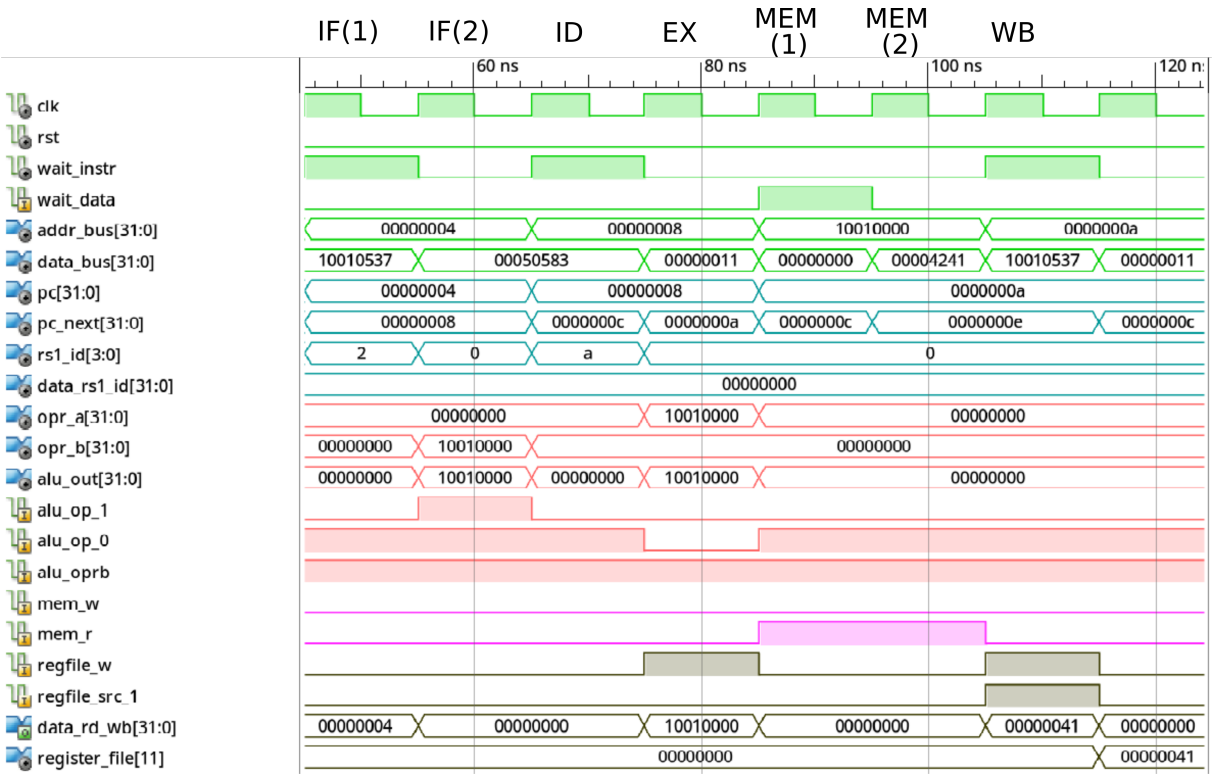
```
section .text:
  PC:

   0: 10010537          lui x10,0x10010
   4: 00050583          lb x11,0(x10) # 10010000 <x_data>
   8: 0001               c.addi x0,0
   a: 0001              c.addi x0,0

section .data:
  PC:

10010000: 4241 x_data
```

A instrução LUI carrega para o registrador x10 o endereço base, que somado ao imediato 0 na instrução LB, forma o endereço de acesso da variável "x_data". Como a instrução carrega apenas um byte do endereço, e no caso, o inferior, o dado armazenado no registrador x11 do banco corresponde a 41. O dado buscado da memória é estendido com zeros para 32 bits. Nota-se que além de dois ciclos de leitura da instrução, se obtém também dois ciclos de leitura de dados no estágio MEM, ambas as memórias possuem um ciclo de latência.

4.2.3.6 Classe de Store

Classe que contém as instruções de escrita da memória: SW, SH e SB. O diagrama da Figura 4.26 demonstra a classe de store em execução dentro do pipeline, o estágio um (IF) é responsável pela leitura da instrução na memória e incremento do PC, sendo o seu incremento transmitido pelos multiplexadores 0 e P para escrita na borda de subida de relógio. O estágio seguinte (ID) realiza a decodificação da instrução, do sinal imediato,

Figura 4.25 – Simulação de uma instrução da classe de load do núcleo pipeline, LB, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.



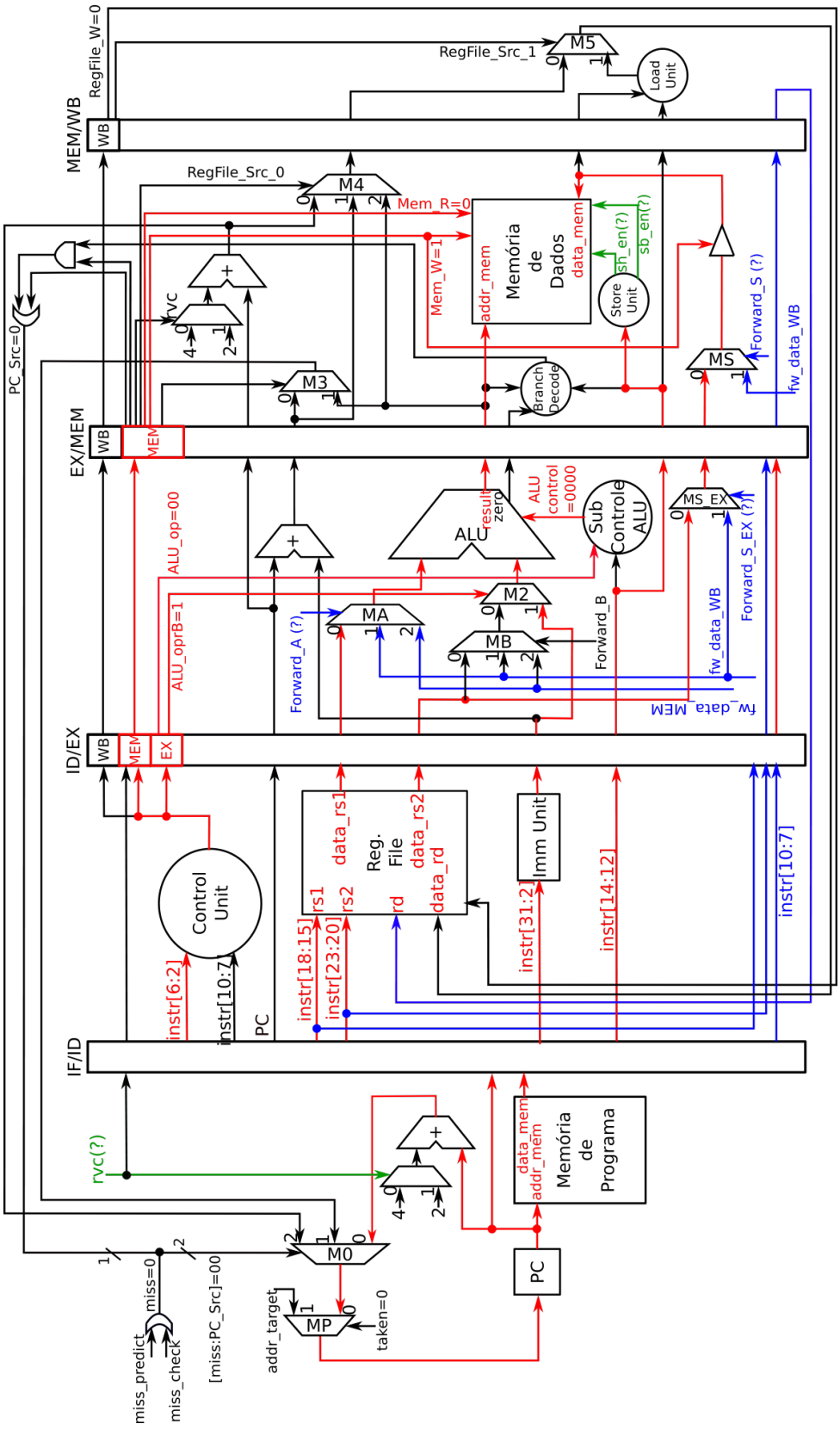
Fonte: Autor.

e também faz a leitura do banco de registradores. O endereço rs1 acessado no banco contém o endereço base para calculo do alvo na memória, e o rs2 o dado que vai ser armazenado. O campo [14:12] da instrução é utilizado para diferenciar o tamanho do dado a ser armazenado (word, half-word ou byte), sendo propagado até o quarto estágio, aonde o mesmo é decodificado na unidade dedicada de store.

O estágio EX possui dois caminhos operando em paralelo, um consiste na passagem da ALU, que realiza a operação de adição conforme indicado pelo sinal "ALU_op=00", com prioridade sobre o campo [14:12] na subunidade de controle da ALU. A soma é feita entre o campo rs1 lido do banco, e o sinal imediato, assim é necessário que o MUX 2 seleciona o mesmo através do nível lógico 1 no sinal de controle "ALU_oprB". O outro caminho é apenas a transferência do dado lido do banco, no campo rs2, para o estágio seguinte, a fim de armazena-lo na memória.

A escrita é indicada para a memória no quarto estágio por meio do sinal "Mem_W" em nível 1, e para evitar conflitos no barramento, se desabilita "Mem_R", colocando o mesmo em nível 0. O sinal "Mem_W" além de ser enviado para a memória externa, também habilita o buffer tri-state do caminho de dados que está conectado ao barramento de

Figura 4.26 – Execução das instruções de escrita na memória.



Fonte: Autor.

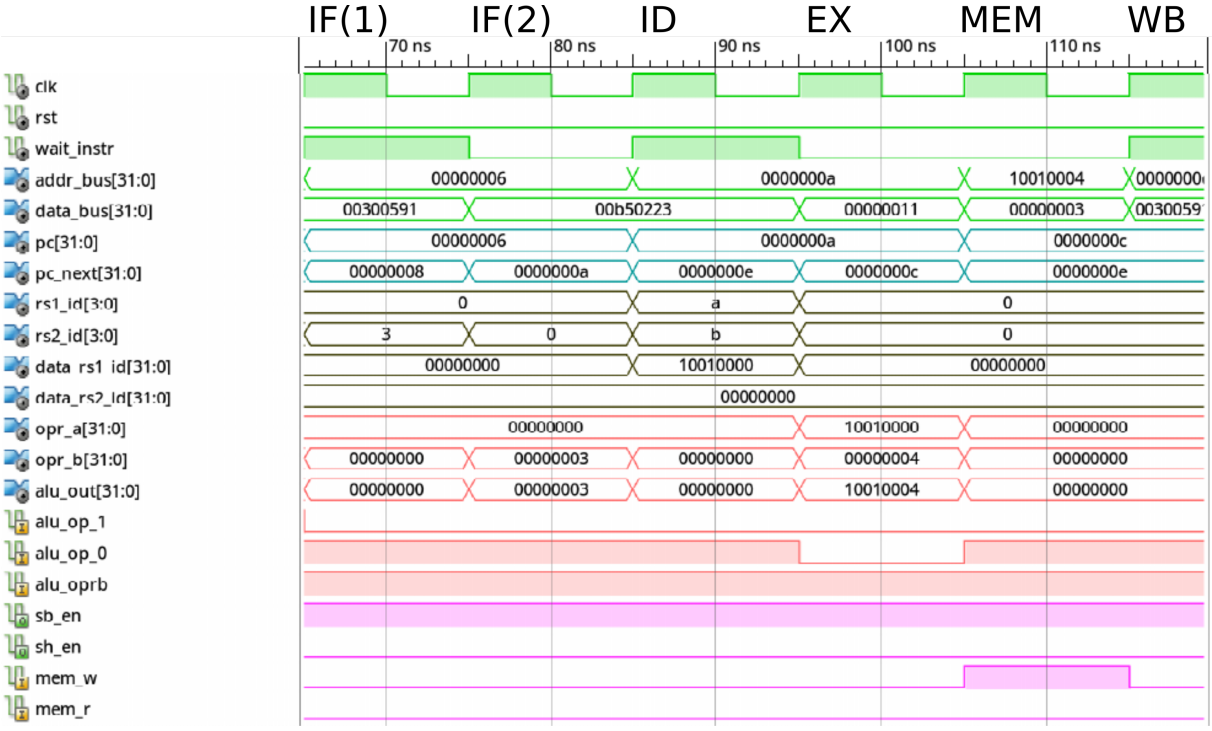
dados do núcleo, pois o mesmo é bidirecional. O campo [14:12] é então utilizado para se enviar ainda mais dois sinais necessários a escrita, "sh_en" e "sb_en", habilitação de escrita por half-word e habilitação de escrita por byte. O quinto estágio não possui relevância, visto que o sinal "RegFile_W" esta desabilitado, nível 0.

Na Figura 4.27 observa-se a simulação de uma instrução SB, em que se armazena o valor decimal 3, presente no registrador x3 do banco, no endereço hexadecimal 10010004 da memória. O código utilizado foi:

```
   PC:

0: 10010537           lui x10,0x10010
4: 458d                c.li x11,3
6: 00b50223           sb x11,4(x10) # 10010004 <_heap_start+0x4>
a: 0001                c.addi x0,0
c: 0001                c.addi x0,0
```

no qual a primeira instrução (LUI) forma o endereço base, que somado ao imediato 4 na instrução SB, forma o endereço alvo.

Figura 4.27 – Simulação de uma instrução da classe de store do núcleo pipeline, SB, com destaque para os estágios do pipeline em que a mesma se encontra, incluindo a latência de acesso a memória. Todos os sinais estão representados na base hexadecimal.



Fonte: Autor.

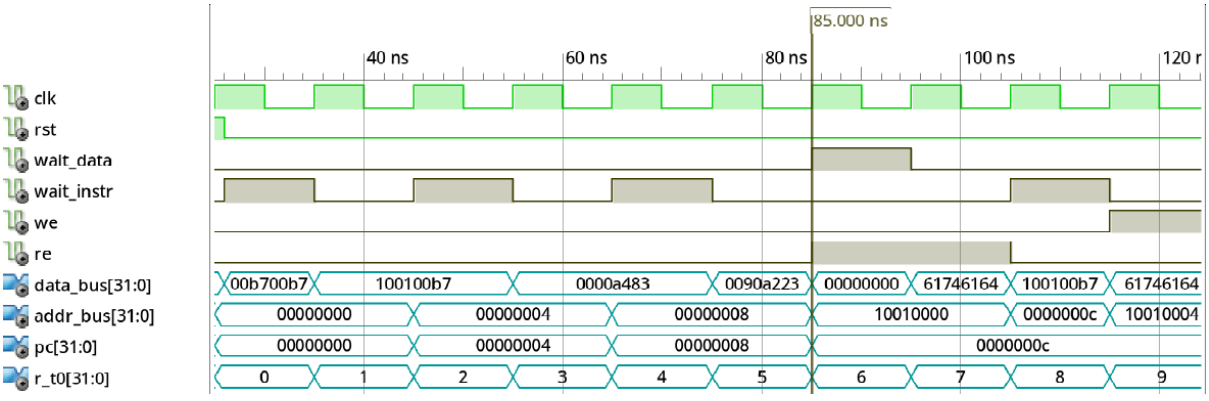
4.2.3.7 Dependências Estruturais

O diagrama de tempo a Figura 4.28 contém a execução de um programa demonstrando a ação dos sinais de espera por instruções e dados da memória, "wait_instr" e "wait_data", respectivamente. O programa executado consiste no seguinte código:

```
end:   codificação     instrução

0:     100100b7        lui x1,0x10010
4:     0000a483        lw  x9,0(x1)
8:     0090a223        sw  x9,4(x1)
c:     0001            c.addi x0,0    # nop
```

Na execução da primeira instrução, logo no primeiro ciclo o sinal "wait_instr" vai para nível lógico 1, pois começou o acesso a memória de programa, e durante esse ciclo o registrador de pipeline entre o estágio ID e IF deve ser atualizado com uma instrução NOP. Assim como todos os elementos de memória do estágio IF são travados por um ciclo, que se resume ao PC e a tabela do preditor. O registrador do temporizador, "r_t0" conta o número de ciclos desde o reset do sistema.

Figura 4.28 – Simulação demonstrando o travamento do pipeline por conta da ação dos sinais de espera pela memória, "wait_instr" para instruções, e "wait_data" para dados. Os sinais "pc", "data_bus" e "addr_bus" estão em hexadecimal, o restante em decimal.



Fonte: Autor.

Após um ciclo de acesso a memória, está disponível no barramento de dados a instrução a ser executada, conforme a codificação do código exemplo, totalizando então, 6 ciclos de relógio para executar uma instrução, considerando o pipeline vazio. Quando se executa uma instrução de load, e a mesma chega no estágio MEM, a prioridade do barramento é transferida para que o acesso seja realizado na memória de dados. Assim, trava-se os estágios anteriores, repetindo por um ciclo a mesma tarefa, no diagrama de

tempo o cursor encontra-se posicionado sobre o inicio do estágio MEM com a instrução load.

O sinal "wait_data" funciona então da mesma forma que o de espera por instrução, durante um ciclo trava-se todo o pipeline, e no próximo habilita novamente seu funcionamento já que o dado desejado está no barramento. A diferença é que mesmo após a disponibilidade do dado no barramento, ainda não se inicializa a leitura da próxima instrução, e sim transfere-se um NOP extra para o estágio ID. Isso também é evidente pela lacuna de dois ciclos extras entre a próxima ação do sinal "wait_instr" quando o load chega no estágio MEM, pois o mesmo espera o load avançar para o estágio final.

Por fim, no caso do acesso a memória RAM para uma escrita, não se tem necessidade de travar o pipeline, já que a escrita irá ser realizada diretamente na memória e um único ciclo de relógio, na borda de subida. O último ciclo de execução demonstrado é quando o store está no estágio MEM, sendo indicado pelo fato do sinal "we" (habilitação de escrita) estar em nível lógico 1 (considerado ativo).

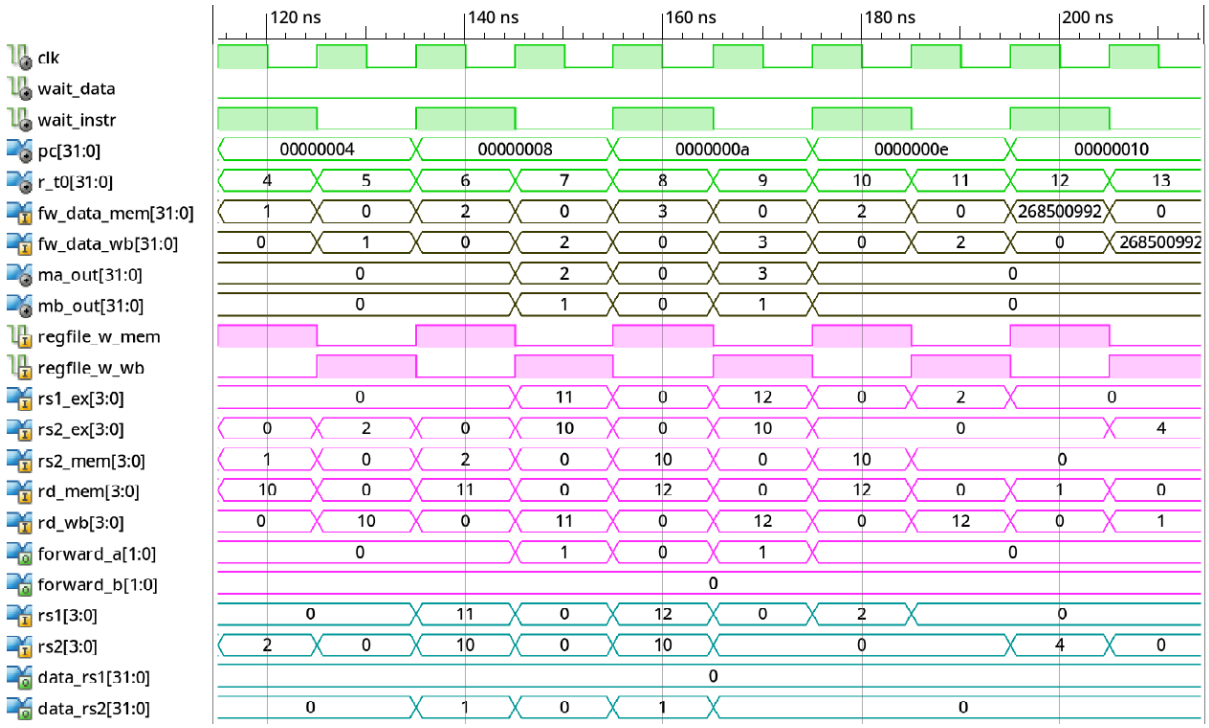
4.2.3.8 Dependências de Dados

A Figura 4.29 apresenta um diagrama de tempo, resultante da execução da técnica de forwarding para resolução de dependências de dados verdadeiras no pipeline. O código executado possuí as seguintes instruções, em ordem:

```
end:  codificação     instrução

0:    4505            c.li  x10,1
2:    4589            c.li  x11,2
4:    00a58633        add   x12,x11,x10    # x12 = x11 + x10
8:    8e09            c.sub x12,10         # x12 = x12 - x10
-----------------------------
a:    100100b7        lui  x1,0x10010
e:    4551            c.li x10,20
10:   00a0a023        sw   x10,0(x1)
14:   00a0a223        sw   x10,4(x1)
```

A simulação começa quando a primeira instrução já está no estágio MEM do pipeline. O sinal "r_t0" consiste no registrador do temporizador, que está contando os ciclos de execução do núcleo, começando de 0. Os sinais com o campo de leitura do banco de registradores (rs1, rs2 e rd) estão rotulados com um sufixo indicando o seu estágio, assim como os sinais dos dados de forward (fw_data), e a habilitação de escrita no banco (regfile_w). Os últimos quatro sinais são as entradas e saídas do banco. Os multiplexadores

Figura 4.29 – Simulação de execução da técnica de forward do pipeline. Com exceção do sinal "pc" que está representado em hexadecimal, os demais estão na base decimal.



Fonte: Autor.

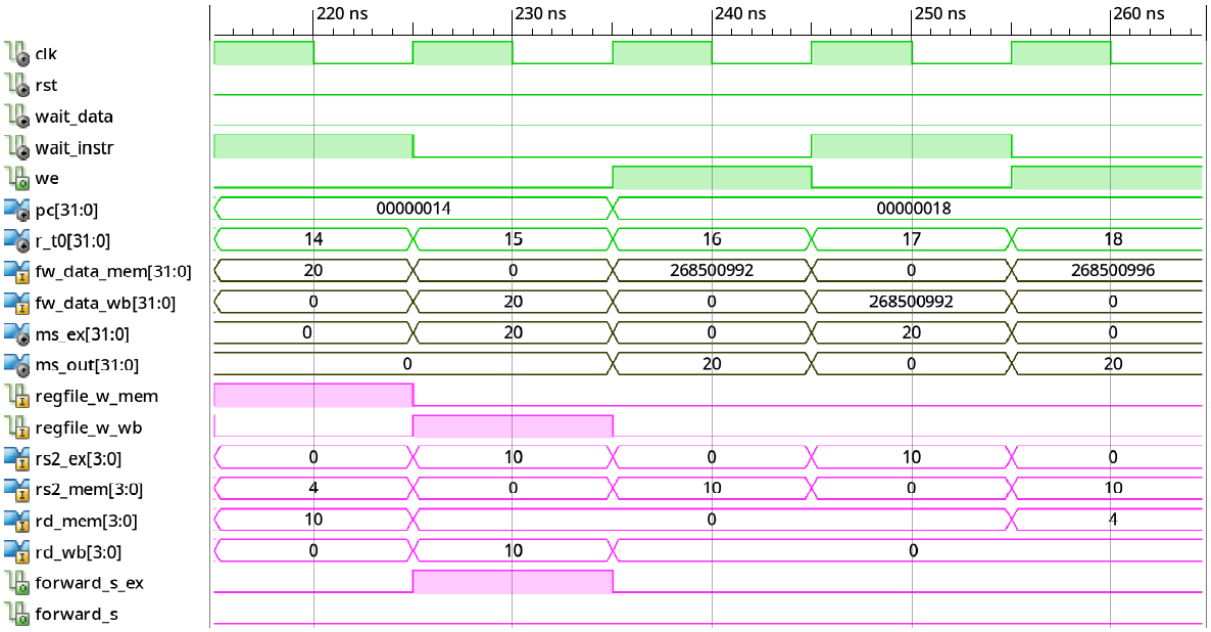
A e B (ma e mb), controlados pela unidade de forward, estão representados pelas suas saídas, para indicar a resolução da técnica, sendo controlados pelos sinais "forward_a" e "forward_b".

De acordo com as dependências criadas no código escrito, se espera que ocorra a realimentação dos estágios WB para o EX, da instrução 1 para a 3, e do estágio MEM para o EX, da instrução 2 para a 3. Com relação a execução do SUB, em teoria se obteria o forward do estágio MEM para o EX. Porém, como observa-se no diagrama de tempo, devido a inserção de uma instrução NOP no estágio ID a cada busca de instrução, se tem sempre uma lacuna de um estágio entre os estágios ativos do pipeline, que estão de fato com código. Logo, qualquer forward que utiliza o caminho entre o estágio MEM e EX, nunca ocorre, como demonstra o sinal "forward_a", que em todos os casos resultou em 1 decimal, selecionando o dado "fw_data_wb", dado do estágio WB. Para selecionar o caminho MEM para EX, seu valor seria 2 (o mesmo se aplica para "forward_b").

A mesma característica de execução do forward pode ser visualizada no diagrama de tempo da Figura 4.30, em que se demonstra a realimentação para instruções de store, durante a execução da segunda parcela do código, instruções 5 a 8. O diagrama começa com a instrução 7, store, já no estágio ID do pipeline. Os multiplexadores controlados pela técnica de forward, nos estágios EX e MEM (ms_ex e ms, respectivamente), respondem

aos sinais "forward_s_ex" e "forward_s". Adicionou-se o sinal de habilitação de escrita da memória (we) para indicar quando as instruções de store estão no estágio MEM, que seria o momento em que o sinal vai a nível 1.

Figura 4.30 – Simulação de execução da técnica de forward do pipeline. Com exceção do sinal "pc" que está representado em hexadecimal, os demais estão na base decimal. Caso de forward para instruções de store.



Fonte: Autor.

O esperado era que da se realizasse o forward do estágio WB para o EX, ligando as instruções 6 e 8, e um forward do estágio WB para o MEM, realimentação do resultado da instrução 6 para a 7. Porém percebe-se que o caminho entre o estágio WB e MEM nunca ocorre, pois da mesma forma que na simulação anterior, se tem uma lacuna de um estágio com um NOP entre as instruções em execução. O store da instrução 8 não chegou nem a necessitar do forward, pois o resultado da instrução 6 já estava armazenada no banco de registradores, quando a mesma fez sua leitura. Dessa forma fica claro que qualquer realimentação entre um estágio, e o seguinte não ocorre devido a característica síncrona do bloco de RAM, com um ciclo de latência na leitura.

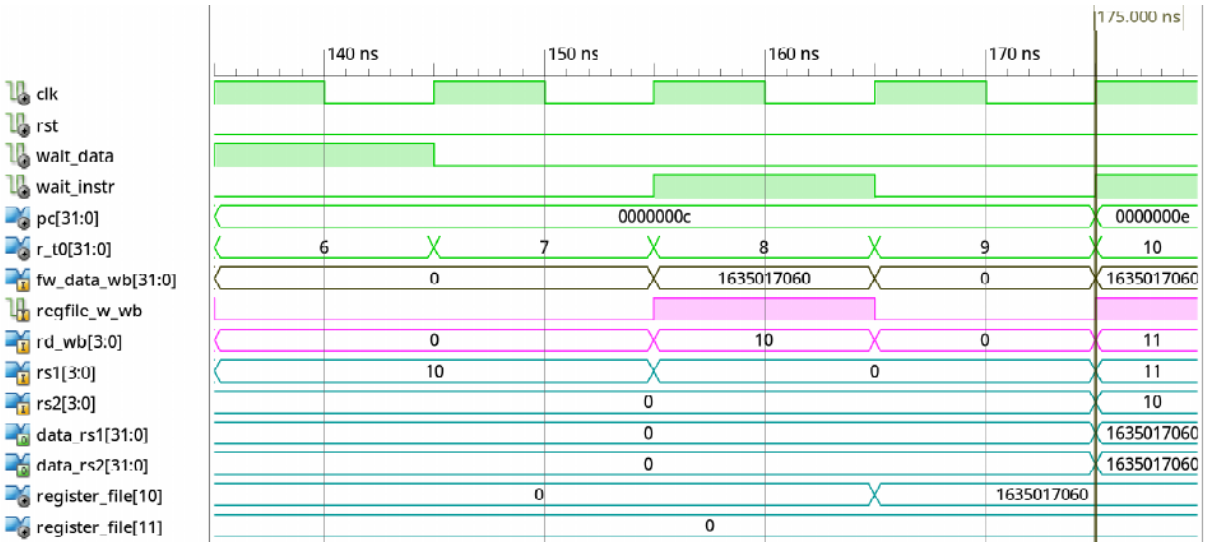
Nota-se também que em ambas as simulações se tem que no momento de escrita, estágio WB, a instrução presente no estágio ID é um NOP inserido pela espera da memória. Assim, o forward interno do banco de registradores também não entrou em ação em nenhuma das situações. Se verifica o forward interno do banco de registradores quando está presente uma instrução de load no pipeline, conforme demonstra a simulação da Figura 4.31, em que se executa a seguinte sequência de instruções:

```
end:  codificação     instrução
```

```
0:      100100b7        lui    x1,0x10010
4:      0000a503        lw     x10,0(x1)
8:      000505b3        add    x11,x10,x0     # x11 = x10 + x0 = x10
c:      95aa            c.add x11,x10         # x11 = x11 + x10
```

com um dado aleatório no endereço a ser acessado da memória. A simulação está representada a partir do ciclo em que a instrução de load está no estágio MEM do pipeline. No diagrama se posicionou o cursor sobre o inicio do ciclo em que ocorre o forward interno do banco de registradores, nesse momento o registrador de leitura "rs1"é o mesmo do destino, "rd_wb". Logo, como se tem também que a escrita no banco está habilitada (regfile_w_wb = 1), o sinal de saída da leitura "data_rs1", é igual ao dado de entrada para escrita do banco (fw_data_wb), e não o valor 0 que o registrador acessado possuí nesse ciclo. Os registradores x10 e x11 do banco são os últimos dois sinais do diagrama de tempo.

Figura 4.31 – Simulação de execução da técnica de forward do pipeline. Com exceção do sinal "pc" que está representado em hexadecimal, os demais estão na base decimal. Caso de forward interno do banco de registradores.



Fonte: Autor.

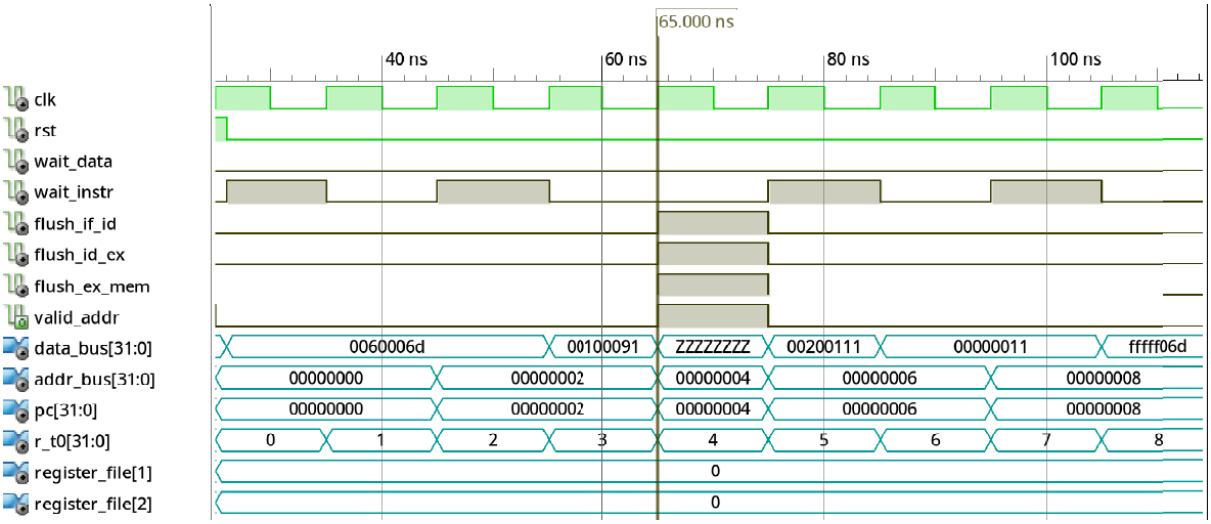
A unidade de hazards posicionada no estágio ID, para detecção de instruções LW seguidas por outras instruções que utilizem seu resultado, também é inutilizada devido a latência de um ciclo para acesso a ROM. Normalmente seria inserido um NOP no estágio anterior ao load, porém, o NOP já está lá desde o primeiro estágio do pipeline, sendo então desnecessária a unidade.

4.2.3.9 Dependências de Controle

O preditor de desvios condicionais dinâmico utilizado no projeto teve sua descrição em VHDL inteiramente de forma inferida, garantindo assim a portabilidade do código. A Figura 4.32 contém uma simulação em que se destaca o uso da lógica de flush do pipeline para desvios incondicionais, de acordo com o código:

```
end:   codificação      instrução

0:     a019             c.j 6
2:     4085             c.li x1,1
4:     4109             c.li x2,2
6:     0001             c.addi x0,0     # nop
```

Figura 4.32 – Simulação do pipeline, demonstrando-se o flush dos registradores de pipeline na ocorrência de desvios incondicionais. Os sinais "pc", "data_bus" e "addr_bus" estão em hexadecimal, o restante em decimal.



Fonte: Autor.

Quando se tem uma instrução de jump sempre ocorre o flush do pipeline, conforme descrito na Seção 3.2.2.6, descartando todas as instruções em execução nos estágios IF, ID e EX, conforme a ação dos sinais "flush" em nível lógico 1 (ativo). Assim, não se escreve nenhum dos valores das instruções 2 e 3 no banco de registradores. Destaca-se que no momento em que a instrução jump está no quarto estágio, MEM, o estágio um está realizando um acesso a memória, e o sinal "wait_instr" sobre condições normais estaria ativo. Dessa forma, ao realizar o desvio incondicional, no próximo ciclo ocorreria a execução da instrução acessada na memória no estágio anterior que é incorreta (pois o PC sofreu um desvio).

O sinal "valid_addr" entra então em nível lógico 1 em conjunto com o sinal de flush no momento do desvio (estágio MEM da instrução jump), para indicar a memória que o acesso deve ser desabilitado. O cursor no diagrama de tempo está posicionado sobre o começo do ciclo em que o flush ocorre, notando também que o barramento de dados (data_bus) fica em estado de alta impedância (símbolo Z), já que nenhuma memória está enviando dados ao barramento, assim como o núcleo.

No diagrama de tempo da Figura 4.33 observa-se o resultado da execução de uma instrução de branch, BEQ (branch if equal), do seguinte código:

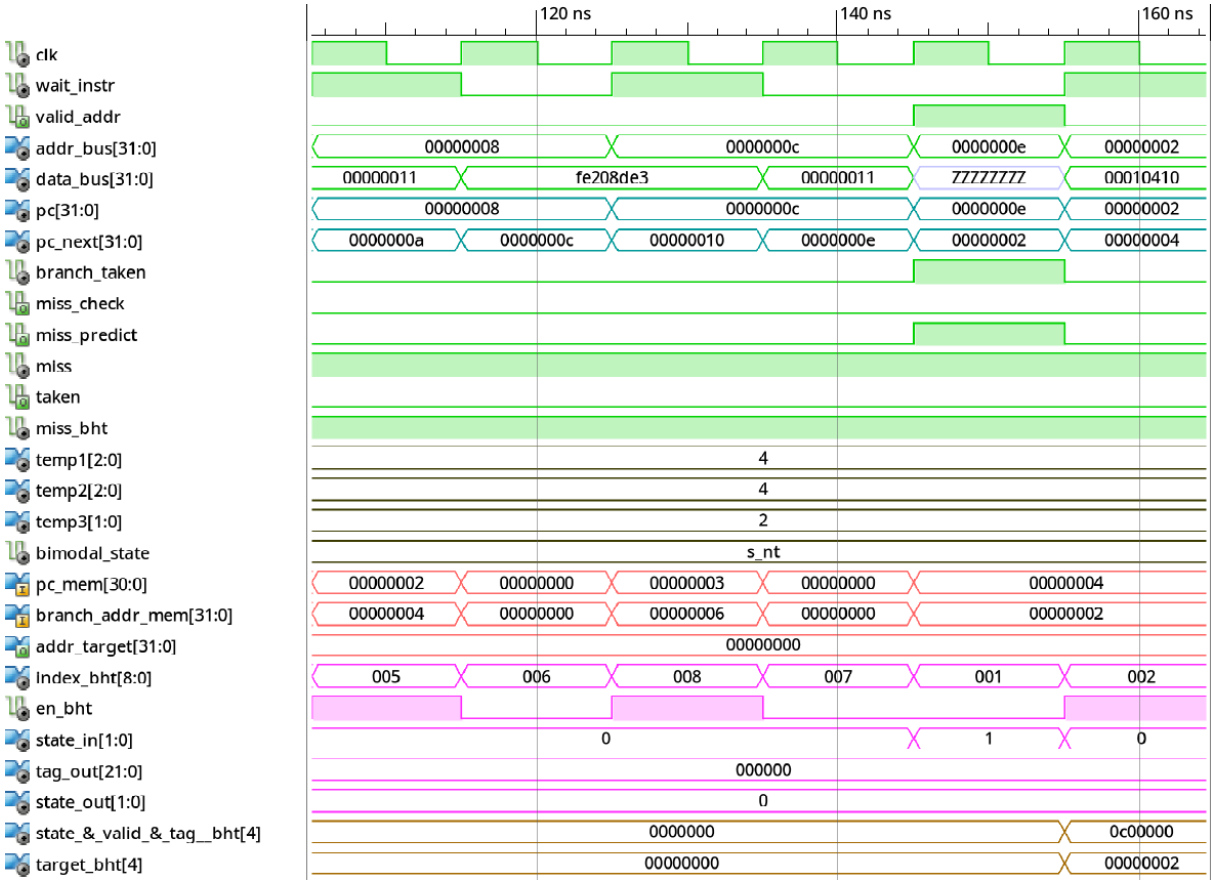
```
   PC:

0: 0001                c.addi x0,0

00000002 <label>:
   2: 0001                 c.addi x0,0
   4: 0001                 c.addi x0,0
   6: 0001                 c.addi x0,0
   8: fe208de3            beq x1,x2,2 <label>
   c: 0001                 c.addi x0,0
```

em que os registradores a serem comparados (x1 e x2) não foram inicializados, logo se forma um loop infinito de desvios para o endereço de "label1" (2 decimal). Os seis ciclos visualizados correspondem a execução completa da instrução BEQ, incluindo a latência da memória no primeiro ciclo, e o estágio WB (ciclo final) que não tem função nenhuma para esta instrução em específico.

No primeiro ciclo de clock é realizada a busca da instrução pelo endereço da saída de PC, conforme indica o sinal "taken" em nível 0 (desabilitado), já que o branch atual não possui entradas na BHT. Caso o preditor tivesse informação a respeito do branch, e fosse dito que o mesmo era not taken, se utilizaria da mesma lógica "taken = 0" para selecionar a saída de PC. No caso, identifica-se que a causa da seleção foi de fato a ausência de informação na BHT devido ao nível ativo do sinal "miss_BHT=1". Quando a instrução se encontra no estágio EX, que pode ser facilmente visualizado pela marcação de 140 ns, é realizada a comparação dos registradores, que no próximo ciclo resulta nos sinais "branch_taken=1". O mesmo indica ao preditor o resultado do branch, que realiza as correções necessárias do núcleo, assim como atualização da BHT.

Para corrigir o caminho de dados (realizar o flush) envia-se o sinal "miss_predict=1" para os registradores de pipeline, trocando as instruções anteriores ao branch que estão em execução por NOP. Atualiza-se então a tabela do preditor com a tag, endereço alvo, estado de predição, e validade do branch em questão. O último sinal, "target_bht[4]", corresponde a entrada da BHT na qual se armazenou o endereço alvo de desvio, sendo

Figura 4.33 – Simulação do pipeline, demonstrando-se o flush dos registradores de pipeline na ocorrência de desvios condicionais. Primeira ocorrência de um branch dentro de um loop infinito. Os sinais estão todos em hexadecimal.



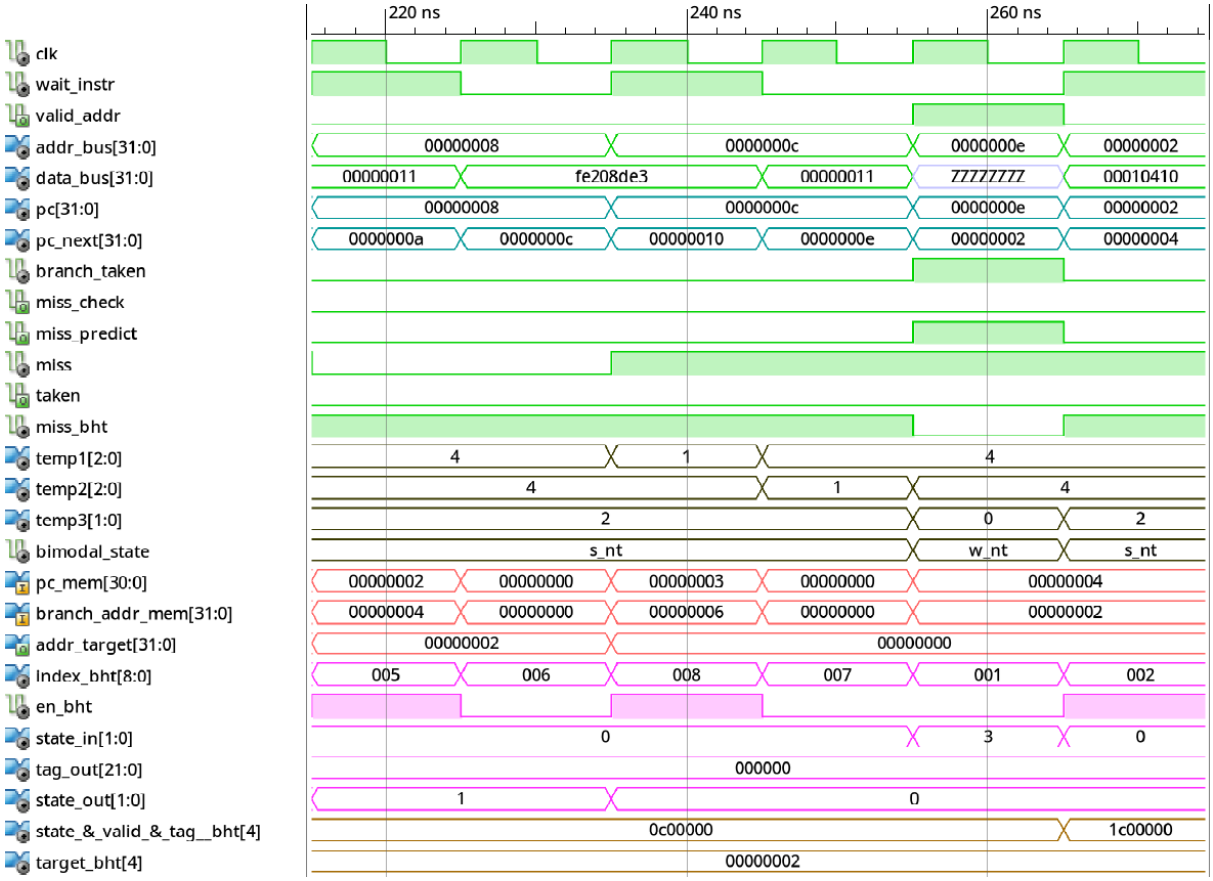
Fonte: Autor.

possível verificar que no último ciclo é o valor da label1, 2 decimal. Anterior ao endereço alvo, encontra-se a entrada com a informação do estado, validade e tag, todos concatenados no sinal "state_&_valid_&_tag__bht[4]". No ciclo final visualiza-se então na entrada a mudança para indicar que agora a informação é valida, e alterar o estado de predição do branch de SNT (strongly not taken) para WNT (weakly not taken), conforme descrito nas Seções 2.3.5 e 3.2.2.6.

A Figura 4.34 contém a segunda execução do branch no loop, na qual agora se obteve informações da BHT, que são propagadas através dos registradores (sinais do diagrama de tempo) "temp1", "temp2" e "temp3", para verificar se a ação SNT foi correta. O sinal "bimodal_state" corresponde a saída da máquina de estados do preditor, sendo o estado atual correspondente ao MSB de "temp3" (no preditor bimodal, o MSB indica a ação). Como o estado corresponde ao armazenado na execução anterior, "w_nt", e o resultado do branch no estágio MEM como taken (branch_taken=1), o sinal "miss_predict" entra em ação, para corrigir o núcleo e atualizar a BHT. A validade, tag, e endereço alvo continuam

os mesmo, modificando-se apenas o estado, que vai para ST.

Figura 4.34 – Simulação do pipeline, demonstrando-se o flush dos registradores de pipeline na ocorrência de desvios condicionais. Segunda ocorrência de um branch dentro de um loop infinito. Os sinais estão todos em hexadecimal.



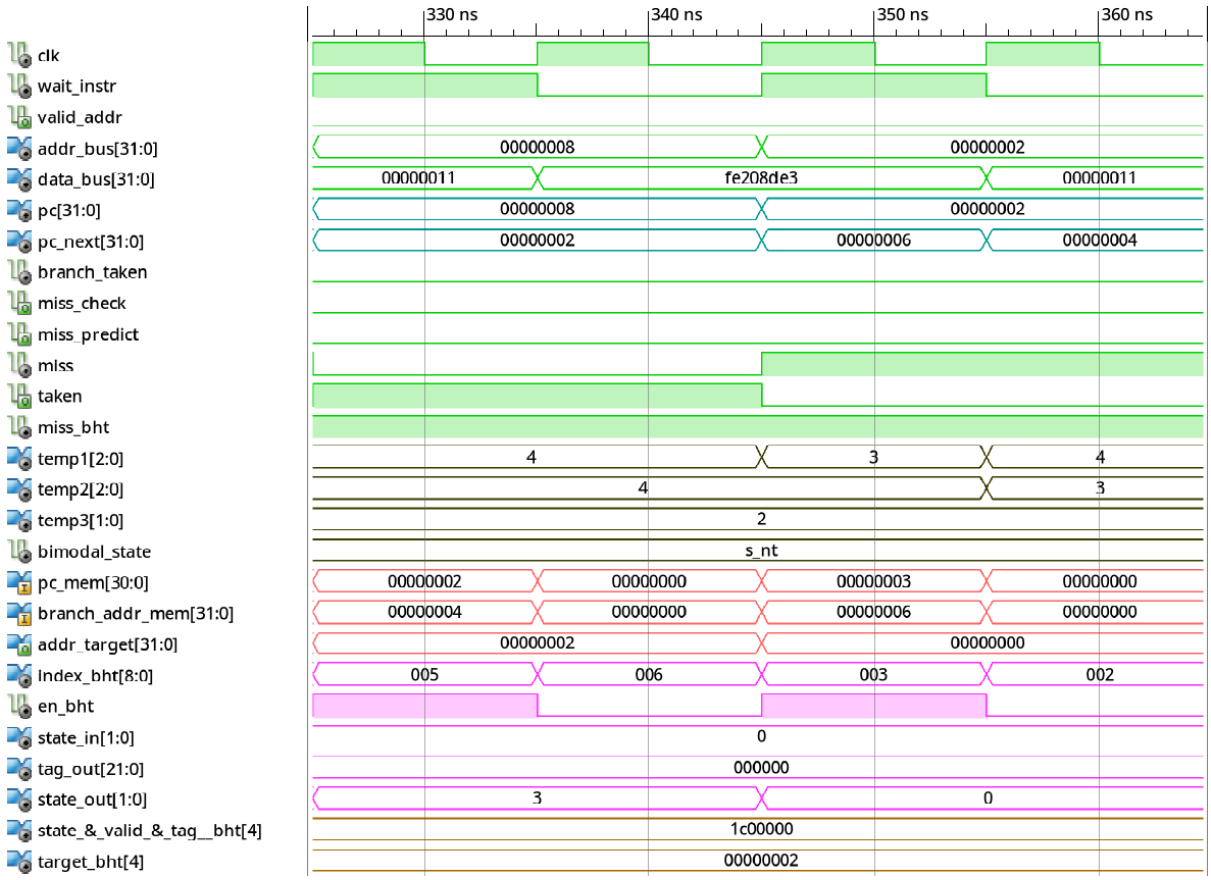
Fonte: Autor.

Por fim, na simulação da Figura 4.35 demonstra-se a próxima execução do branch, no qual agora o preditor acerta a ação ao desviar para o endereço 2, logo após a busca da instrução BEQ. Conforme o sinal "taken", o próximo valor de PC (pc_next) após o branch, e consequente uso para busca de instrução, corresponde ao valor do sinal "addr_target" dos primeiros dois ciclos.

4.3 MAPEAMENTO DA MEMÓRIA

As descrições em VHDL dos componentes de memória utilizado, para ambos os núcleos, estão disponíveis para visualização no Apêndice C. Todas as memórias foram descritas de forma inferida, de acordo com as especificações da Seção 3.3, utilizando os blocos de RAM da FPGA Spartan-6, conforme o guia de HDL por Xilinx (2009b, p. 135).

Figura 4.35 – Simulação do pipeline, demonstrando-se o flush dos registradores de pipeline na ocorrência de desvios condicionais. Terceira ocorrência de um branch dentro de um loop infinito. Os sinais estão todos em hexadecimal.



Fonte: Autor.

4.3.1 Script de Link

Para a escrita de programas com o mapeamento de memória desejado, se criou um arquivo de link, que continha o mínimo de informações necessárias sobre quais endereços de memória atribuir para cada instrução gerada do código em C ou assembly, no processo de compilação, ligação e montagem com as ferramentas GCC. Devido a limitação da quantidade de BRAM disponível, e também utilizadas no projeto, alterou-se o endereço inicial do segmento stack, ao disponível para uso, conforme discutido na Seção 3.3.1. O script de link escrito está disponível no Apêndice A.1.

4.3.2 Leitura de Instrução

A leitura de instruções ocorre da mesma forma para ambos os núcleos, visto que a memória ROM utilizada é a mesma. A diferença está presente no fato de que, apesar de a

ROM resultar em 32 bits de dados em uma operação de leitura, no multicíclico apenas se ignora os 16 bits superiores internamente no núcleo.

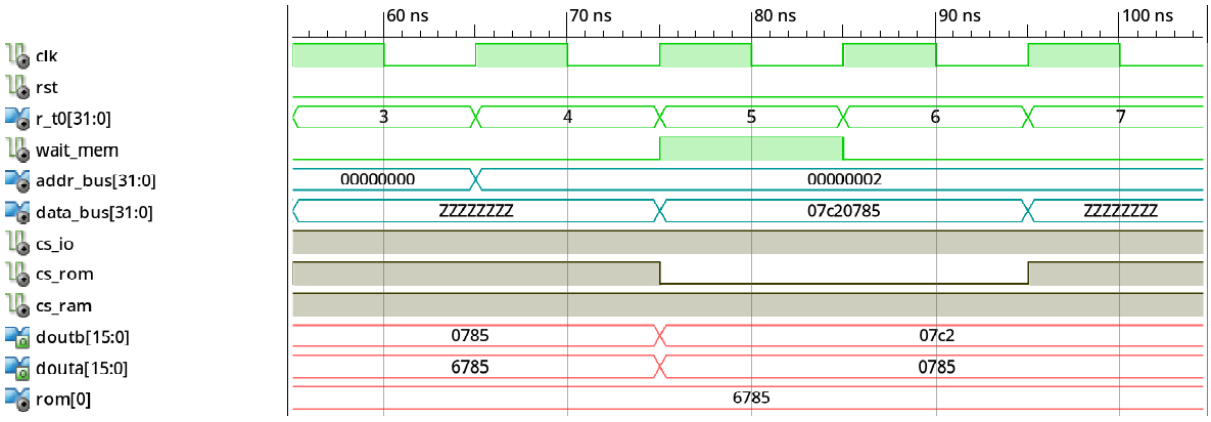
4.3.2.1 Simulação - Núcleo Multicíclico

Visualiza-se na simulação da Figura 4.36 a leitura de uma instrução de 16 bits da extensão C do RISC-V, das seguintes instruções armazenadas na memória:

```
addr:

0: 6785                 c.lui x15,0x1
2: 0785                 c.addi x15,1
4: 07c2                 c.slli x15,0x10
```

No primeiro e segundo ciclo do diagrama a instrução c.lui está sendo finalizada, atualizando-se então o valor do PC no núcleo multicíclico. Como o mesmo não realiza constantemente a leitura de instruções da memória, se coloca em alta impedância o barramento de dados e sinaliza-se para o decodificador não começar a contagem de 1 ciclo de latência do sinal "wait_data".

Figura 4.36 – Simulação da leitura na memória ROM de uma instrução de 16 bits c.ADDI. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

No ciclo seguinte então se realiza normalmente o acesso a ROM com um ciclo de exposição da instrução ao barramento, e outro para leitura do mesmo, sendo a instrução disponível no barramento de dados durante o quarto ciclo a c.addi. Os 2 bytes superiores do barramento são ignorados, correspondentes a porta B (doutb), pois correspondem a outra instrução, a necessidade de duas portas se da por conta da leitura de dados constantes de 32 bits da região de memória da ROM.

Percebe-se que na leitura da instrução c.addi a mesma já estava disponível logo no ciclo em que se estava esperando pela latência da memória. Ocorrência devido ao fato de a leitura da memória não ser desabilitada, apesar de o barramento e a lógica do sinal "wait_mem" serem. O não aproveitamento dessa característica para evitar o stall de um ciclo de acesso é proveniente das instruções de desvio, as mesmas escrevem o endereço alvo do PC apenas no último ciclo, logo não se tem um acesso constante a ROM ocorrendo durante sua execução para busca da próxima instrução.

4.3.2.2 Simulação - Núcleo Pipeline

O diagrama de tempo da Figura 4.37 demonstra a leitura de duas instruções a serem executadas no núcleo pipeline, uma da base E (32 bits) e outra da extensão C (16 bits), conforme:

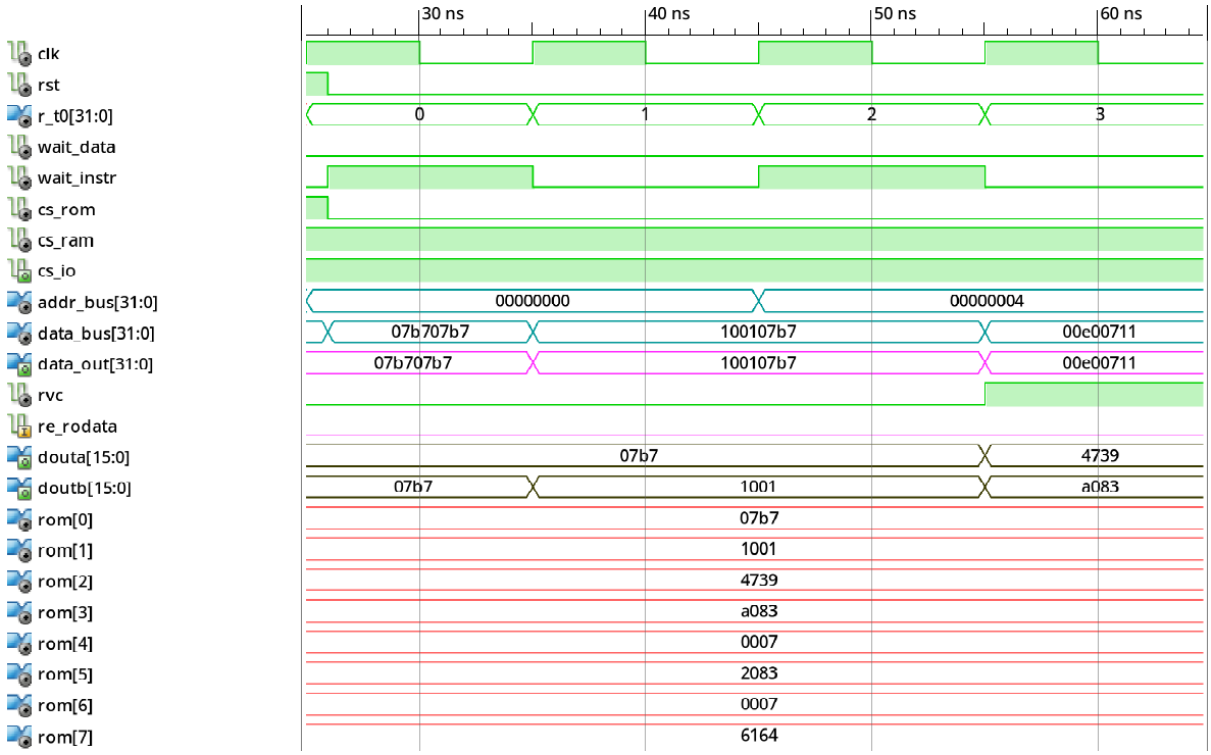
```
 addr:

0: 100107b7           lui x15,0x10010
4: 4739                c.li x14,14
```

O registrador do temporizador, "r_t0", é utilizado como referência em conjunto com o sinal de clock, no ciclo 0 se inicializa a leitura da instrução LUI de 32 bits. Ao receber o endereço do barramento que foi identificado como da seção de programa, se habilita o sinal "cs_rom" (em nível 0), e por consequência se ativa o sinal "wait_instr" por um ciclo, os demais sinais de habilitação (da RAM e de disp. IO) permanecem desabilitados em nível 1. No ciclo 1 já é possível realizar a leitura da instrução disponível no barramento de dados, "data_bus", que é igual ao valor do sinal "data_out", saída do decodificador RVC.

Os dados das portas da ROM, "doutb" e "douta" (porta B e A, respectivamente) passam primeiramente pela unidade de decodificação RVC, porém, como o sinal "rvc" esta desabilitado apenas se ignora essa etapa. Transfere-se então diretamente para a saída, "data_out", a instrução resultante da leitura. Ao iniciar-se o ciclo 2, uma nova leitura é realizada na ROM a fim de buscar novamente uma instrução, sendo que no quarto ciclo o decodificador RVC sinaliza que a mesma é de 16 bits, através do nível lógico 1 no sinal "rvc". Dessa forma, percebe-se que o valor presente no barramento (e também no sinal de saída do decodificador RVC) difere da instrução armazenada na memória (c.li). Pois a mesma foi estendida para a sua versão correspondente de 32 bits.

O diagrama de blocos da Figura 4.38 demonstra essa diferença no valor entre a instrução presente no barramento, e a armazenada na memória, assim como a codificação de ambas. A equivalente da instrução RVC c.LI é a ADDI, utilizando como um dos operando o registrador x0 (constante 0), no campo do registrador fonte rs1. O outro operando é o

Figura 4.37 – Simulação da leitura na memória ROM de uma instrução de 32 bits LUI, seguida por uma de 16 bits c.LI. Todos os sinais estão representados em hexadecimal.



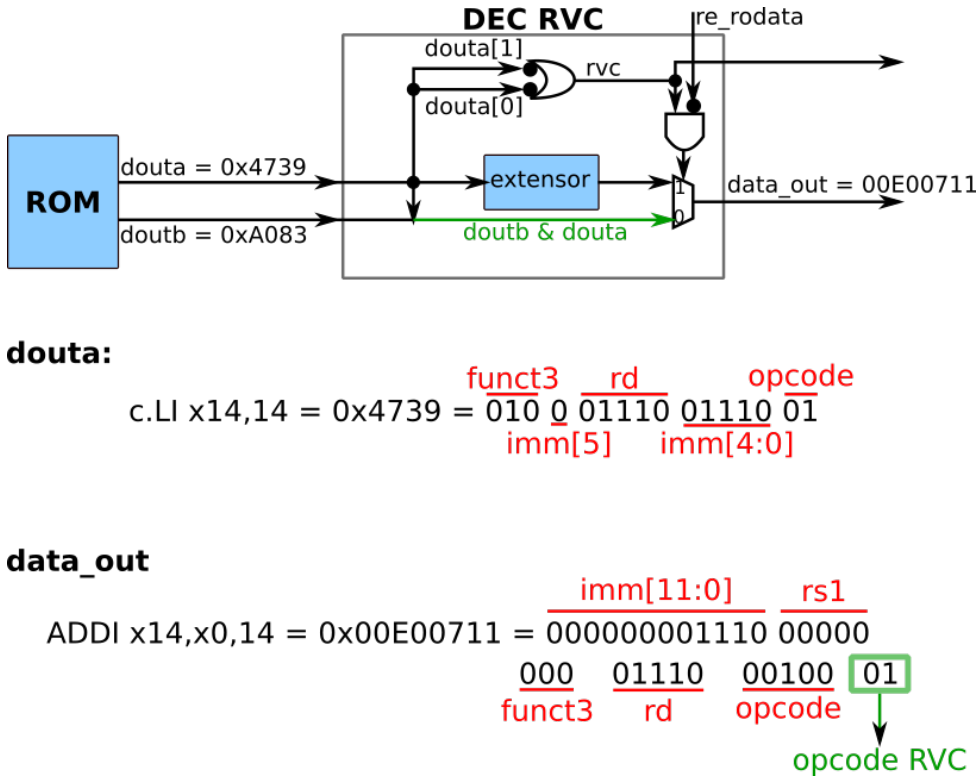
Fonte: Autor.

imediato de 6 bits, que é estendido pelo MSB para 12 bits, e o registrador destino (x14), correspondente ao campo rd. Os campos de decodificação diferem das instruções RVC para as de 32 bits, logo, tanto funct3 quanto opcode são diferentes, ressaltando que os 2 LSB do opcode para instruções de 32 bits são de nível 11. Porém, como os mesmos só atribuem informação para a decodificação RVC (já utilizada nesse momento) não faz diferença o valor que será atribuído a eles, já que o núcleo não irá utilizar para nenhuma finalidade Observa-se que a extensão também é ignorada em outro caso além das instruções de 32 bits, quando se realiza uma leitura de dados constantes (seção rodata da memória) é necessário ignorar essa etapa, a fim de evitar a leitura errada de dados. O dado da porta B é ignorado quando a instrução detectada é de 16 bits.

4.3.3 Leitura de Dados

A leitura de dados da memória pode ocorrer tanto na memória ROM, quanto na RAM, sendo que o decodificador de controle da memória utilizado em conjunto ao núcleo multicíclico não oferece suporte de leitura de byte ou half-word, apelas palavras completas (32 bits).

Figura 4.38 – Detecção e decodificação de uma instrução de 16 bits da extensão C, para sua equivalente de 32 bits da base E do RISC-V.



Fonte: Autor.

4.3.3.1 Simulação - Núcleo Multicíclico

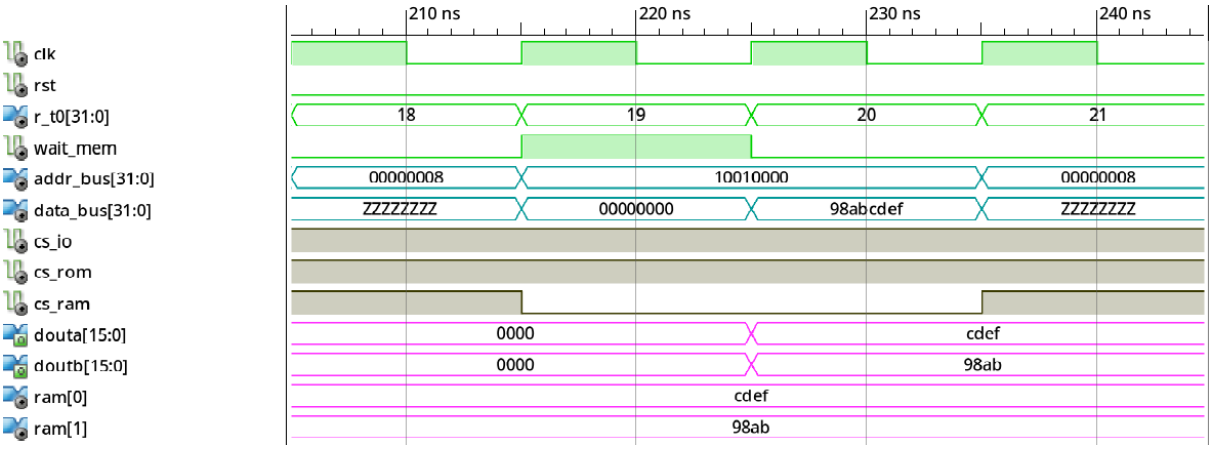
A simulação do diagrama de tempo da Figura 4.39 contém a simulação de um acesso a memória RAM por meio da instrução LW executada no núcleo multicíclico. A sequência de instruções executada foi:

```
 addr:

section .text

   0: 6785                 c.lui x15,0x1
   2: 0785                 c.addi x15,1
   4: 07c2                 c.slli x15,0x10
   6: 4398                 c.lw x14,0(x15)
   8: 4018                 c.lw x14,0(x8)

section .data:

10010000 <_data>:
```

```
10010000: 98abcdef
```

No ciclo 18, indicado pelo sinal "r_t0", se calcula o endereço de acesso a memória, 0x10010000, sendo então utilizado no ciclo 19. Para que o núcleo trave sua execução enquanto espera pela disponibilidade do dado desejado no barramento, o decodificador da memória coloca em nível 1 o sinal "wait_mem" por um ciclo de relógio, sendo somente desabilitado no ciclo seguinte. Nesse momento o dado desejado, indicado pelos sinais "ram[0]" e "ram[1]", pode ser lido do barramento de dados, sendo os seus valores visualizados também nas portas de saída A e B da RAM, "douta"e "doutb"respectivamente. Durante todo o intervalo de acesso a memória se habilita a RAM pelo sinal "cs_ram=0".

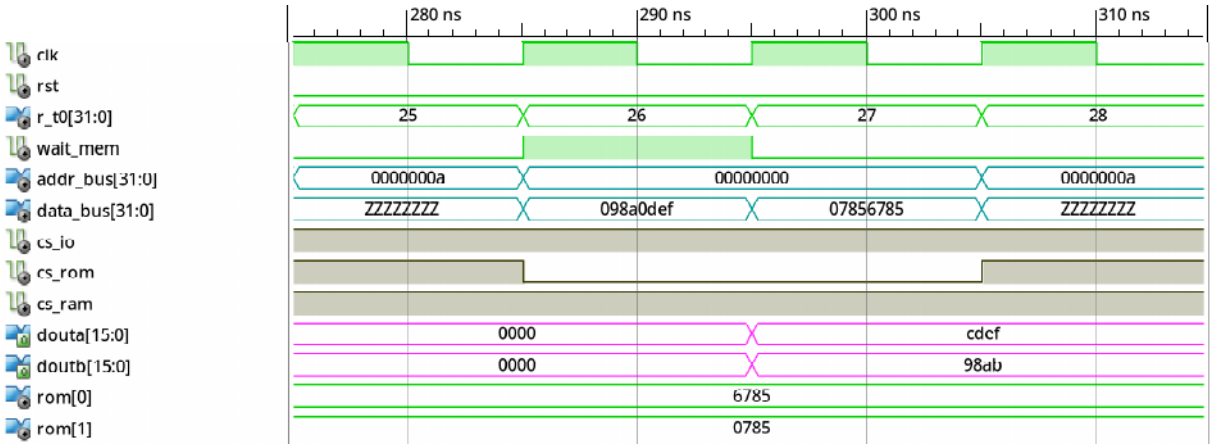
Figura 4.39 – Simulação de leitura de dados da RAM, por meio da instrução LW. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

O segundo load word executado tem como alvo de acesso a memória ROM, visto que tanto o imediato utilizado, quanto o endereço base são nulos (registrador x8 não inicializado). Por inspeção dos valores contidos nos endereços das instruções do código sabe-se que o valor que se desejar ler é 0x07856785 (as duas primeiras instruções concatenadas), conforme se demonstra na simulação da Figura 4.40. Acessando agora a memória ROM para leitura de dados, se realiza a busca de um dado constante (rodata), que se optou por reutilizar o valor das primeiras instruções para fins de demonstração da operação. No ciclo 26 se inicia o acesso a memória, colocando-se em nível 1 o sinal de espera pela memória, "wait_mem". Após um ciclo se tem então disponível no barramento o dado desejado para leitura, nota-se que dessa vez habilitou-se a ROM (cs_rom = 0), e não a RAM, do contrário iriamos ler novamente os mesmos dados da operação anterior, presentes nas portas A e B da RAM ("douta"e "doutb").

Figura 4.40 – Simulação de leitura de dados da ROM, por meio da instrução LW. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

4.3.3.2 Simulação - Núcleo Pipeline

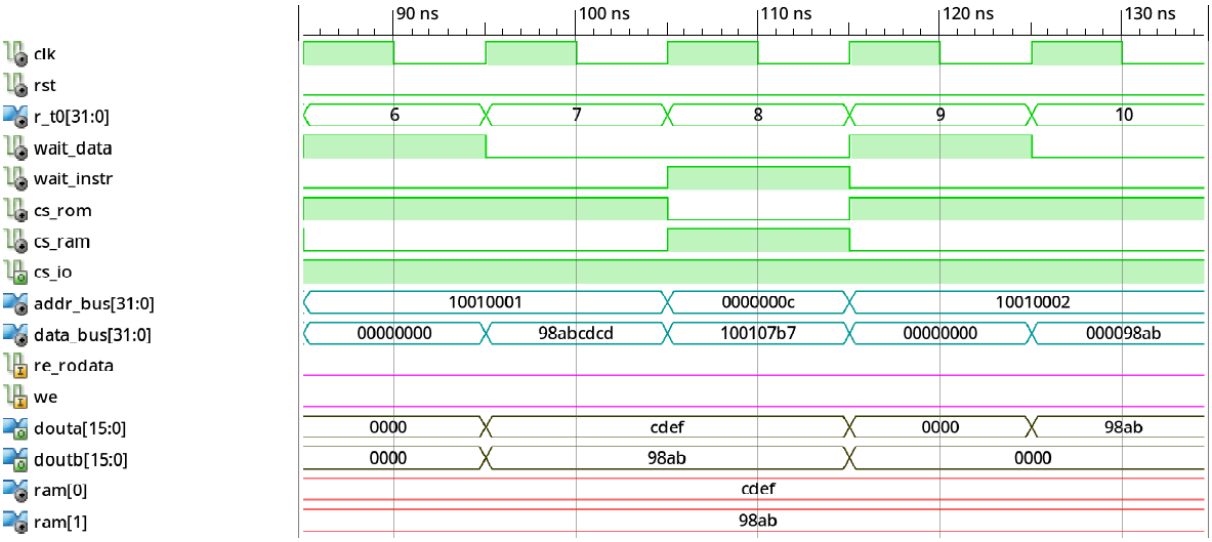
O diagrama de tempo da Figura 4.41 apresente a leitura de dados da RAM, executada pela instrução de acesso a byte LB. O valor do temporizador, sinal "r_t0" é utilizado como referência de ciclo de relógio na simulação, começando pelo ciclo 6 em que a instrução LB do código:

```
 addr:

section .text:

   0: 100107b7          lui x15,0x10010
   4: 00178083          lb x1,1(x15) # 10010001 <_data+0x1>
   8: 0027d083          lhu x1,2(x15) # 10010002 <_data+0x2>

section .data:

10010000 <_data>:
10010000: 98abcdef
```

chegou ao estágio MEM. Nesse momento o decodificador detecta pelo endereço do barramento que o acesso é a RAM, e habilita o mesmo (cs_RAM = 0), enquanto desabilita os demais (cs_ROM e cs_IO). Devido a latência de um ciclo de acesso, se coloca o sinal "wait_data" em nível 1, para dizer ao núcleo que o dado no barramento não está pronto, estando o mesmo disponível apenas no ciclo 7. Nesse momento o dado do endereço acessado, 0x10010001, está disponível na porta B, com os 16 MSB, e na porta A com os 16

LSB. Os sinais correspondentes as portas A e B, são respectivamente "douta" e "doutb".

Figura 4.41 – Simulação de leitura de dados da RAM. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

O byte que se deseja ler é o 0xcd, porém o mesmo deve estar na posição menos significativa do resultado de leitura, logo, se identifica através do LSB do endereço no barramento qual dos bytes da palavra se deseja ler, no caso, byte 1. Atribui-se então ao barramento de dados do sistema o valor da porta B e A (multiplexado o byte 1 para o lugar do 0), aonde percebe-se que o dado 0xcd está na posição desejada. Internamente o núcleo irá estender o sinal do dado para 32 bits, partindo do bit 7, logo, ignora-se os demais campos dentro da unidade de decodificação da memória. No ciclo 9 se realiza novamente um acesso para leitura de dados, dessa vez por meio da instrução LHU, leitura de meia palavra unsigned. Como a memória está divida por meia palavra não é necessária nenhuma lógica extra para identificar os bytes que se deseja ler, basta ignorar a porta B da mesma forma que na instrução LB. No quinto estágio do pipeline o dado irá ser estendido com 0 para 32 bits. Sendo assim, uma instrução LH[U] opera da mesma forma que uma LW internamente no decodificador da memória.

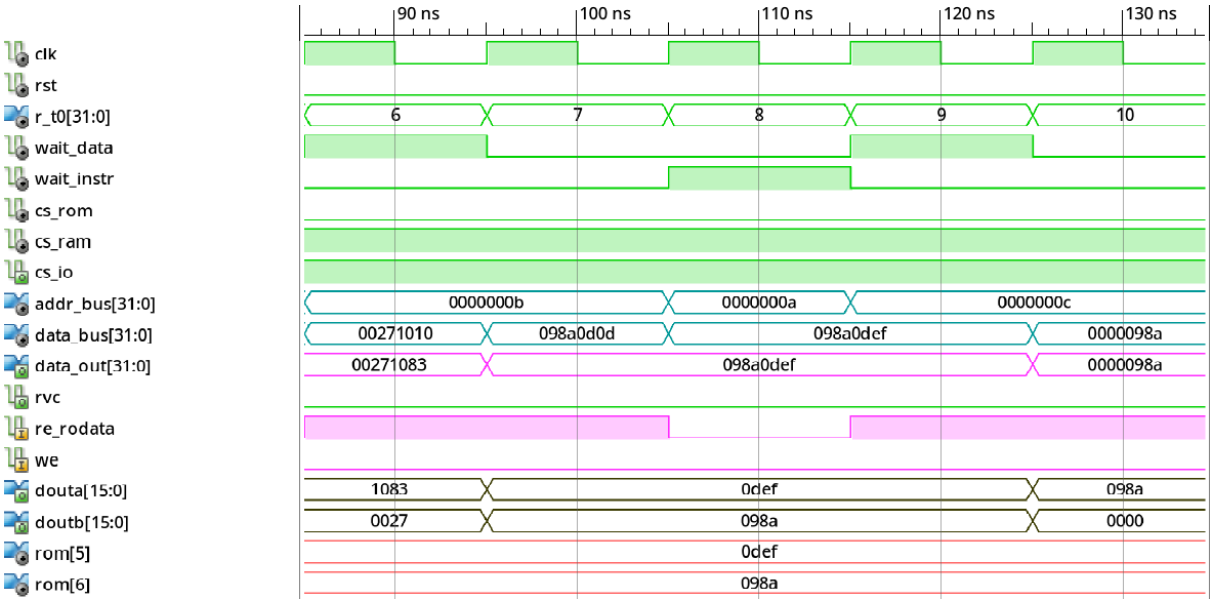
4.3.3.3 Simulação RO Data - Núcleo Pipeline

Para a leitura de dados RO (read-only), o acesso acontece na ROM, funcionando de acordo com a Seção 4.3.2.2, porém com uso do sinal "re_rodata". O diagrama de tempo da Figura 4.42 contém a simulação de leitura para duas instruções de load na ROM, LBU e LH, conforme o código:

```
 addr:

section .text

0: 4729                 c.li x14,10
2: 00174083             lbu x1,0(x14)
6: 00271083             lh x1,2(x14)

 0000000a <_rodata>:
a: 098a0def
```

No ciclo 6, indicado pelo valor do registrador do temporizador (r_t0), se inicia a leitura do dado, como a leitura é destina a dados se ativa o sinal "wait_data" e não "wait_instr", mesmo que o acesso seja a ROM. Pois os sinais possuem diferentes efeitos internamente no núcleo.

Figura 4.42 – Simulação de leitura de dados da ROM. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

Também percebe-se que ativou-se o sinal "re_rodata", agora em nível lógico 1 para leitura de dados constantes, ação necessária devido ao fato de os dados de saída da ROM (porta B e A, sinais "doutb" e "douta", respectivamente) passarem pelo decodificador RVC. Logo se utiliza esse sinal para ignorar tal ação que pode alterar os dados. O resultado está contido no sinal "data_out", que é conectado então ao barramento de dados principal, "data_bus". No ciclo 7 está disponível então no barramento o byte de leitura alvo, que no

caso seria 0xef, os demais bits superiores no barramento podem ser ignorados, o núcleo irá estender o sinal a partir do bit desejado (8) com zeros.

No ciclo 9 inicia-se então novamente um acesso para leitura de dado constante, dessa vez acessando apenas meia palavra, tendo como alvo os 2 byte superior do dado armazenado na memória, 0x098a. De acordo com os 2 LSB da meia palavra acessada, o decodificador entraria em ação, e causaria modificações não desejadas na leitura, porém, visualiza pelo sinal do barramento no ciclo 10 que o mesmo ficou inalterado conforme indicado ao decodificador RVC que era necessário, pelo sinal "re_rodata". O sinal "rvc" também deve ser mantido em nível lógico 0 (inativo) além de ignorar-se a decodificação de instruções RVC, pois o mesmo é enviado ao núcleo.

4.3.4 Escrita de Dados

A escrita de dados da memória ocorre unicamente na RAM, possuindo diferenças significativas entre sua implementação para uso com o núcleo multicíclico, e o núcleo pipeline, por exemplo, habilitação de escrita por byte para uso com as instruções SB e SH, ausentes no núcleo multicíclico.

4.3.4.1 Simulação - Núcleo Multicíclico

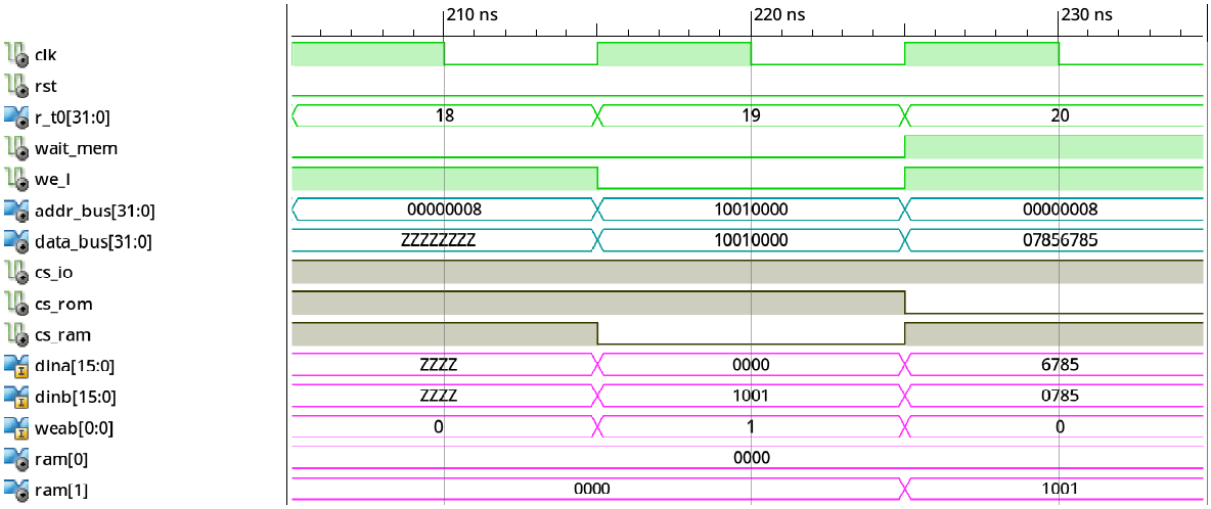
Visualiza-se no diagrama de tempo da Figura 4.43 a simulação de uma instrução SW. No ciclo 18, indicado pelo valor atual do registrador "r_t0" (temporizador), o núcleo multicíclico está calculando o endereço de acesso da RAM, que será o primeiro de sua região da memória, 0x10010000. Em sequência, no ciclo 19, realiza-se o acesso a memória para escrita, indicado pelo sinal "we_l", ativo em nível 0, que por consequência de seu valor, e do decodificador da memória identificar a região de acesso pelo endereço no barramento, resulta no valor 0 para o sinal "cs_ram" (ativo). Os sinais de entrada da porta A e B da RAM correspondem respectivamente a "dina" e "dinb", que compartilham do mesmo sinal de habilitação de escrita ativo em nível 1, "weab".

Nesse instante o dado no barramento a ser escrito é 0x10010000, que aparece armazenado na RAM no ciclo 20, nos sinais "ram[0]" e "ram[1]", para a menos e mais significativa do dado, respectivamente. O código executado foi:

```
addr:

0: 6785                 c.lui x15,0x1
2: 0785                 c.addi x15,1
4: 07c2                 c.slli x15,0x10
```

```
6: c39c                 c.sw x15,0(x15)
```

Figura 4.43 – Simulação de escrita na RAM, por meio da instrução SW. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

4.3.4.2 Simulação - Núcleo Pipeline

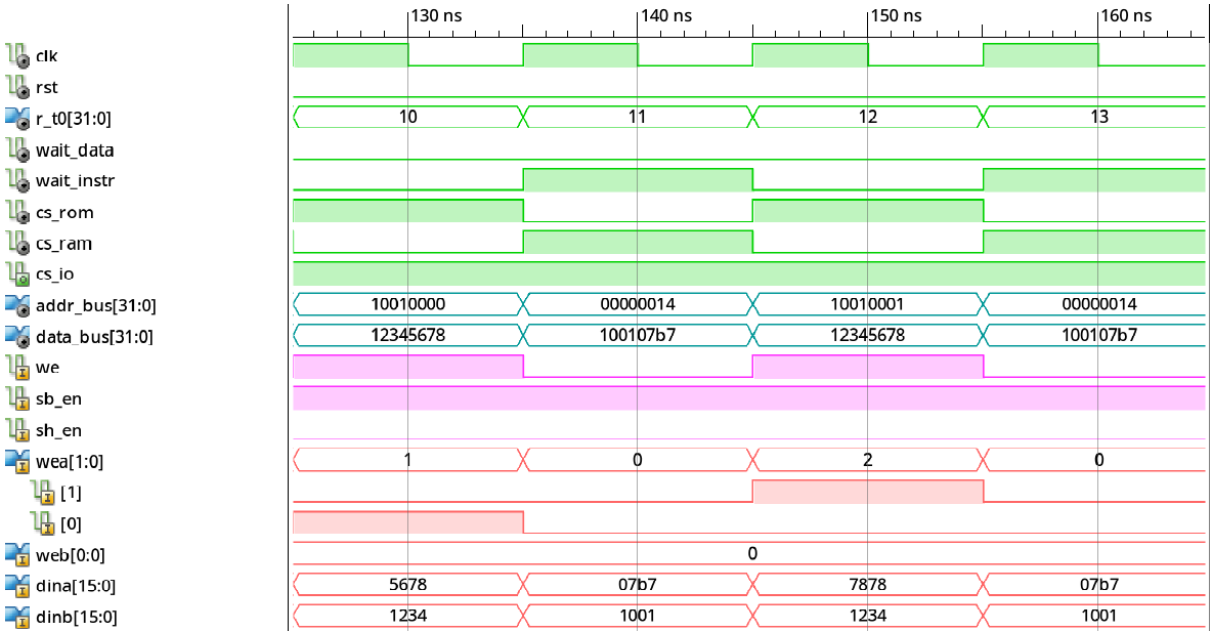
O diagrama de tempo da Figura 4.44 contém a simulação da operação de escrita da instrução SB, executada duas vezes, para o byte inferior e superior do endereço da memória. O código executado foi:

```
 addr:

0: 100107b7           lui x15,0x10010
4: 123452b7           lui x5,0x12345
8: 67828293           addi x5,x5,1656 # 12345678 <_stack_start+0x2331678>
c: 00578023           sb x5,0(x15) # 10010000 <_data>
   10: 005780a3           sb x5,1(x15)
   14: 00579123           sh x5,2(x15)
   18: 0057a223           sw x5,4(x15)
```

No bloco de RAM dual port da memória RAM, apenas a porta A é utilizada para acessos que não são de 32 bits, logo o sinal "web" da porta B é de um bit, enquanto o sinal de habilitação de escrita da porta A, "wea" possui dois bits. O LSB se refere ao byte inferior do endereço, enquanto que o MSB ao superior. Percebe-se na simulação que na execução

do primeiro store, quando o mesmo se encontra no estágio MEM, que o registrador do temporizador (r_t0) esta contando o décimo ciclo, apenas o bit 0 de "wea" foi ativado. Assim se escreveu apenas o byte desejado da memória. Os sinais de controle do núcleo que indicam quando a escrita é de byte ou half-word são, respectivamente "sb_en" e "sh_en", sendo que "we" é a habilitação geral de escrita da memória.

Figura 4.44 – Simulação de escrita de dados na RAM, por meio da instrução SB. Todos os sinais estão representados em hexadecimal.



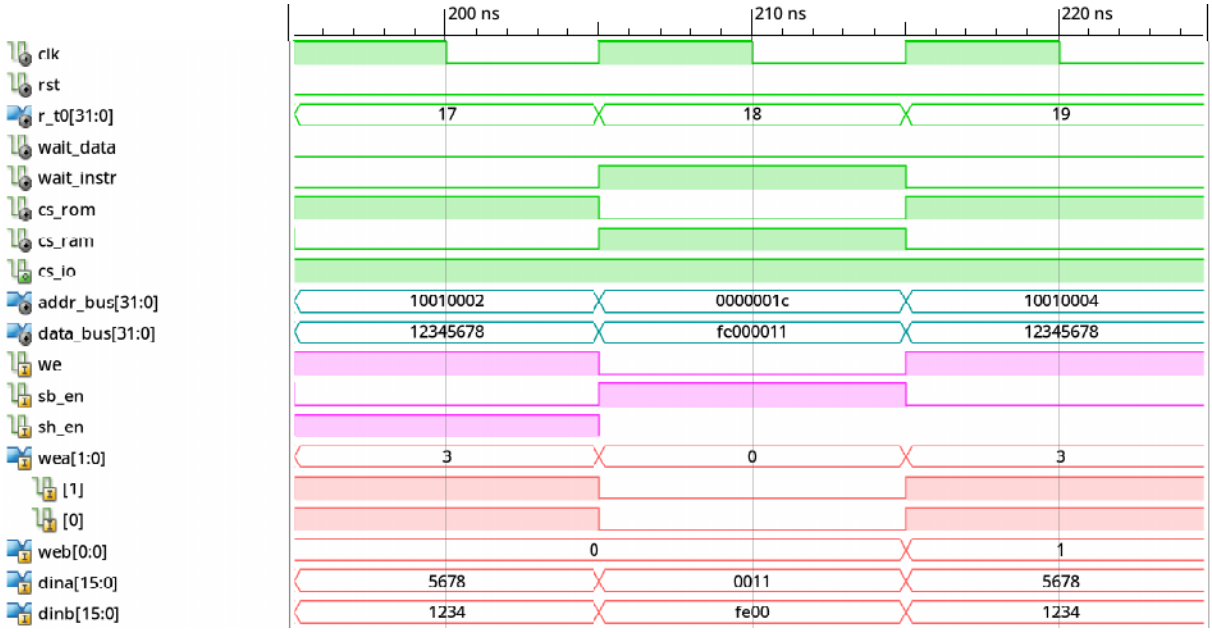
Fonte: Autor.

O simulador utilizado (ISim) não possui suporte para visualização da forma de onda de variáveis, logo não foi possível demonstrar no diagrama de tempo o dado armazenado na mesma. No ciclo 12 do diagrama se executa o segundo store byte, dessa vez se deseja escrever apenas no byte superior, do mesmo endereço, resultando então no bit 1 de "wea" indo para nível lógico 1, enquanto que o restante das habilitações de escrita em nível 0. O dado a ser armazenado do barramento, em instruções SB e SH, é sempre a porção menos significativa, sendo necessário então multiplexar os 2 bytes inferiores do barramento para os 2 superiores da porta A, como é possível de se visualizar no valor do sinal "dina", que corresponde ao dado de entrada da porta A, "dinb" sendo a porta B.

No diagrama de tempo da Figura 4.45 é demonstrado por fim a simulação das instruções SH e SW, do mesmo código. Ao executar a operação de escrita de meia palavra no ciclo 17, é necessário habilita ambos os bytes para escrita na porta A, logo se obtém "wea = 11", deixando a escrita da porta B inativa. Como os acessos a memória são alinhados, não é necessário nenhum meio de multiplexação para trocar os bytes do barramento de posição para essa instrução (SH). No ciclo 19 tem-se então a instrução SW, sua única di-

ferença em relação a SH é o valor de "web", que agora esta habilitado, completando então 32 bits a serem escritos na memória de dados.

Figura 4.45 – Simulação de escrita de dados na RAM, por meio das instruções SH e SW. Todos os sinais estão representados em hexadecimal.



Fonte: Autor.

4.4 E/S

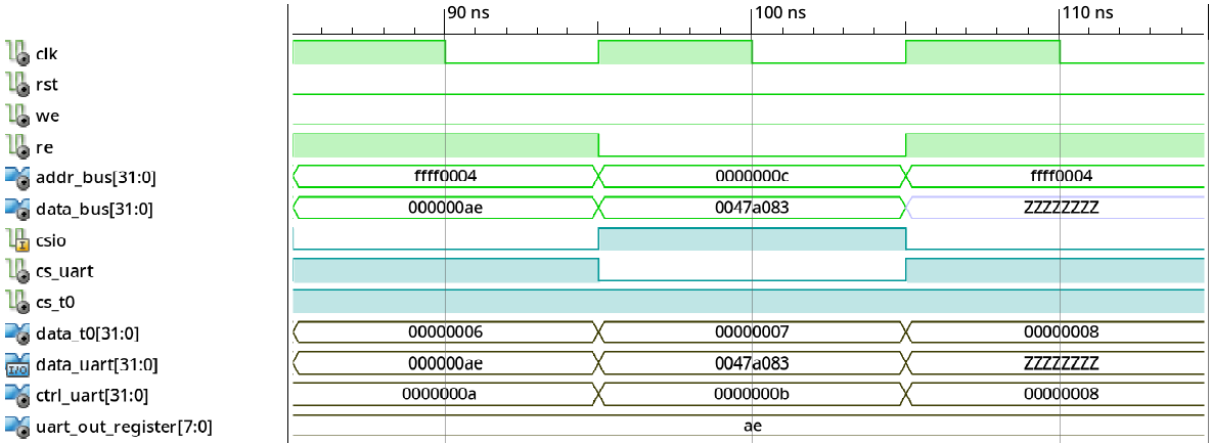
Está disponibilizado no Apêndice D a descrição VHDL dos componentes E/S utilizados, de acordo com as especificações da Seção 3.4.

4.4.1 UART - Leitura

A simulação da Figura 4.46 demonstra a execução de leitura do dispositivo E/S UART, executando uma instrução de acesso a memória no núcleo pipeline, sendo o seu comportamento idêntico para o núcleo multicíclico. Dos três ciclos de relógio visualizados no diagrama de tempo, o primeiro e o terceiro caracterizam dois acessos de leitura, tanto que o sinal de habilitação de leitura (re) está em nível ativo em ambos. No barramento de endereços, "addr_bus", se tem disponível o alvo de leitura, que corresponde ao registrador de dados da UART, e no barramento de dados (data_bus) o resultado da leitura que foi estendido com zeros para 32 bits (UART envia e recebe dados de 8 bits).

O dado a ser lido na primeira instrução de load, corresponde ao armazenado no registrador de saída da UART, sinal "uart_out_register", que é transmitido ao barramento de acordo com as habilitações dos sinais em nível ativo, "csio=0" e "cs_uart=1" ("csio" e "cs_t0" são ativos em 0, e "cs_uart" em 1). Após a primeira leitura o registrador de controle da UART muda de valor, "ctrl_uart", indicando que o próximo dado ainda não está disponível. Resultando então na atribuição de estado alta impedância ao barramento de dados na tentativa de leitura nova da UART, do terceiro ciclo de clock, visto que o mesmo não tem nenhuma informação para enviar.

Figura 4.46 – Simulação da leitura de dados recebidos no dispositivo E/S UART, por meio da instrução LW do núcleo pipeline. Todos os sinais estão na base hexadecimal.



Fonte: Autor.

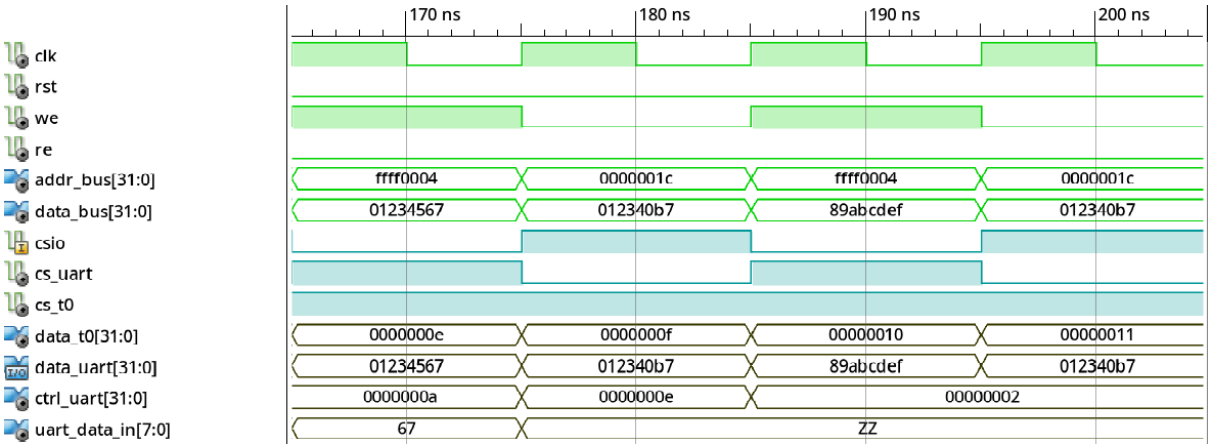
4.4.2 UART - Escrita

Visualiza-se no diagrama de tempo da Figura 4.47 uma operação de escrita na UART, no qual o primeiro ciclo de relógio representa o momento em que uma instrução SW entrou no estágio MEM. Nesse momento ela acessa então o endereço da UART (addr_bus), e escreve o dado no barramento, em seu registrador temporário de dados, visto que a transmissão de dados é lenta e necessita do valor por tempo prolongado. O sinal "uart_data_in" corresponde a porta da entrada de dados da UART, que sendo de apenas 8 bits, transmite apenas a parcela menos significativa do dado presente no barramento. Para que a escrita do dado presente na porta de entrada de dados da UART ocorra, os sinais de habilitação "csio" e "cs_uart" devem estar em seus níveis ativos, 0 e 1, respectivamente. Evidentemente, o temporizador deve estar desabilitado a fim de evitar conflitos de dados no barramento.

No segundo ciclo o registrador de controle da UART muda de valor (ctrl_uart), para

que seja possível a verificação do estado de envio do dispositivo por meio da técnica de polling, também nota-se que a porta de entrada fica em estado de alta impedância, pois o valor já está armazenado dentro do registrador temporário. No caso da simulação demonstrada, o terceiro ciclo de relógio representa outra instrução SW, armazenando um dado diferente na UART, que resultaria em 0xb7 (porção menos significativa do barramento). Porém, como não se verificou o estado da UART (intencionalmente), a escrita continua desabilitada e a porta de entrada permanecerá em alta impedância.

Figura 4.47 – Simulação da escrita de dados no dispositivo E/S UART, por meio da instrução SW do núcleo pipeline. Todos os sinais estão na base hexadecimal.



Fonte: Autor.

5 CONCLUSÕES E PERSPECTIVAS

Neste trabalho verificou-se o uso da ISA do RISC-V para o projeto de um núcleo multicíclico capaz de executar a extensão C do RISC-V, e um pipeline executando tanto a extensão C quanto a base E. Como uma das características presentes no RISC-V sendo a flexibilidade de escolha dos tipos de instruções que se deseja implementar, utilizou-se apenas da extensão C no núcleo multicíclico, por mais que a mesma não tenha sido criada com a finalidade de uso sem uma das bases. Mesmo assim, implementou-se a característica da base E, a limitação do número de registradores de uso geral, obtendo-se assim então um núcleo com número mínimo de registradores e tamanho de instrução (16 bits) do RISC-V. O seu funcionamento foi comprovado por meio de diversas simulações para as diferentes classes de instruções, assim como testes práticos em que se verificou o funcionamento correto do processador. Tendo em vista que os dados manipulados são de 32 bits (apesar das instruções compressas de 16), o núcleo não obtém um ganho relevante de área, o que condiz com a especificação da extensão C, que busca reduzir o tamanho do código, ação importante em sistemas com pouca memória disponível.

O uso da extensão C sozinha também influenciou no número de sinais de controle, como no banco de registradores por exemplo, em que se necessita de sinais extras para controlar o acesso limitado ao mesmo, que é uma das características que possibilita a compressão de instruções. Assim como também diversas instruções que utilizam um operando do banco não especificado na instrução, como o stack pointer (x2). Ou então endereços de escrita não do banco que não estão contidos no campo destinado ao mesmo da instrução, como em instruções de jump. Na etapa de decodificação da instrução se tem a necessidade de que a mesma ocorra de forma rápida, devido a esta necessidade de controlar a leitura do banco, da mesma forma que se deseja decodificar o sinal imediato embaralhado no sinal. Essas duas unidades que operam em paralelo levaram a decisão da implementação de uma máquina de estados de mealy sobre a unidade de controle, a fim de se obter uma resposta mais rápida. O processador multicíclico possui como limitação a escrita de programas unicamente em assembly, pois a extensão C não foi criada como um conjunto de ISA para uso sozinho, e o compilador não possui opções de utilizar somente a mesma. Ao programar em ASM utiliza-se apenas as instruções então desejadas (RVC) do RISC-V.

Na implementação do núcleo pipeline procurou-se utilizar das técnicas mais simples para lidar com os principais problemas que o mesmo propõe, os hazards. Para a dependência de dados no caso, a única presente era do tipo verdadeira (RAW), sendo resolvida pela realimentação do forward, que consiste em um dos caminhos mais longos dentro do núcleo. Passando por diversos multiplexadores, e no caso do estágio EX, da ALU, que pode ocorrer de executar uma operação relativamente longa, como deslocamentos lógicos. Sendo um dos problemas perceptíveis nos resultados e simulações, que dos totais

de 6 caminhos possíveis de forwarding, apenas 3 ocorrem, mais especificamente para instruções que estão separadas por um estágio de pipeline. Essa característica que surgiu por conta da latência de acesso a memória utilizada no projeto, de um ciclo que acaba por inutilizar um dos principais caminhos do núcleo, sendo a mesma lógica aplicada a unidade de hazards localizada no estágio ID, que detecta hazards de instruções de load seguidas por uma reutilizando seu resultado.

O preditor de branch funcionou corretamente, sendo capaz de se adaptar aos desvios condicionais conforme os mesmo ocorriam, evitando perdas de ciclo em diversas situações. Seu tamanho ocupou o 2 de 32 BRAMs disponíveis na FPGA, utilizando-se uma para o endereço alvo e uma para o restante das informações (bit de validade, tag e PC do branch). Após a inicialização do sistema, percebeu-se que o mesmo também não era possível de reiniciar suas entradas, pois como característica os blocos de RAM tem a ausência de um reset geral. A implementação do processador pipeline possui possibilidade de escrita de programas tanto em ASM quanto em C, porém o suporte a base E ainda não é pleno na versão 8.2 do RISC-V GCC, logo não foi utilizada com intensidade a linguagem C. Verificou-se também no núcleo pipeline as perdas de ciclo causadas pelo uso de um barramento de dados e endereço compartilhado para ambas as memórias, de programas e dados. Sendo a prioridade do barramento para a memória de dados, quando a mesma era acessada acabava por travar todo o núcleo nos estágios anteriores.

A opção de se utilizar um decodificador externo ao núcleo RV32EC para instruções da extensão C diminuiu a complexidade do núcleo pipeline, pois o mesmo não precisa se adaptar a tantas instruções a serem executadas, mantendo o foco nas implementadas da base E. Ficando então por conta do decodificador dedicado a extensão C a expansão da instrução de 16 bits, para sua equivalente de 32 bits. Decodificador esse que se alocou na saída da memória de programas, deixando assim qualquer relação a instruções compressas fora do hardware do núcleo. A única lógica presente no núcleo pipeline com respeito a instruções RVC diz respeito ao PC, que deve ter seu incremento controlado entre 2 ou 4.

Na implementação do banco de registradores de ambos os núcleos se utilizou de LUTs, essa opção considerou as limitações que o uso de uma BRAM acarretavam. Ao utiliza uma BRAM, se iria desperdiçar uma grande quantia de espaço da mesma, pois o banco não preenchia nem metade do dispositivo, e o uso dessa parte vazia iria significar acréscimo de controle. Também tem como ponto fundamental a questão do número de portas, que nos blocos de RAM é de 2, e no banco desejou-se 3, duas de leitura, e uma de escrita. O banco de registradores estava ligado diretamente ao uso da base E do RISC-V escolhida, logo, caso se opta-se por utilizar blocos de RAM também perderia o significado da redução de 32 para 16 registradores de uso geral, devido a grande quantia de armazenamento que o bloco oferece.

A utilização da memória resumiu-se ao suficiente para o teste das instruções, apesar de ter sido projetada com certa profundidade a fim de utilizar-se dos blocos de RAM

da FPGA. Seu tamanho da memória de programa e dados limitou a possibilidade dos programas a serem executados, assim como também o fato de que no presente momento do trabalho, o compilador não oferecia suporte pleno a base E do RISC-V. Mesmo assim, foi possível por meio da execução de um programa teste em ASM comprovar o funcionamento de ambos os núcleos, visualizando o resultado no computador pessoal por meio do dispositivo UART. A forma que se utilizou para testar os núcleos na prática mostrou-se um dos principais problemas no decorrer do projeto, pois a mesma era de baixa confiabilidade (dependia de um certo grau de funcionamento prévio do sistema), e ineficiente visualmente do ponto de vista do usuário (projetista). Todos os componentes de memória obtidos foram descritos em VHDL, forma inferida, garantindo assim a portabilidade do código.

Em todos os casos obteve-se o mínimo de performance desejado na síntese/implementação do hardware, que se resume ao uso do oscilador de 100 MHz presente na placa Nexys 3. Porém percebe-se que, no caso do pipeline com mais intensidade, o resultado foi próximo de ficar abaixo, consequência de decisões como a ausência de uso de diversos recursos presentes na FPGA, visto que aproveitou-se no presente trabalho basicamente das BRAMs. O número de LUTs ocupadas não passou da metade em nenhum dos processadores, deixando margem para expansões no hardware, seja do núcleo, ou de outros componentes.

5.1 SUGESTÕES PARA TRABALHOS FUTUROS

Para trabalhos futuros, propõem-se possíveis melhorias na performance do hardware obtido para o núcleo pipeline, por meio de modificações possíveis de acordo com os resultados, como:

• Estudar a possibilidade de reduzir a queda de desempenho devido a latência de acesso aos blocos de RAM;

• Possibilidade de alteração na organização da BHT utilizada no preditor de desvios, a fim de melhorar sua performance, já que a implementação atual não possui nenhum algoritmo de substituição;

• Estudo da exceções e interrupções, evitando assim o grande desperdício de tempo visualizado no uso de periféricos como a UART.

• Um melhor aproveitamento dos recursos da FPGA Spartan-6 para implementação de componentes específicos do núcleo, melhorando o aproveitamento de lógica e velocidade (clock) do mesmo.

Do elemento de memória:

• Transportar a memória principal do sistema para fora do dispositivo FPGA, pois o mesmo não possui armazenamento suficiente para programas mais complexos, sendo uma possibilidade o uso das memórias disponíveis na placa Nexys 3;

• Adaptação do circuito/descrição do componente de memória utilizado, para uma memória cache, aproveitando melhor os recursos de RAM disponíveis na FPGA.

Em relação a programação e testes do hardware:

• Procurar por outras formas de realizar testes práticos do sistema, que não envolvam métodos não confiáveis como unicamente a UART, como por exemplo, uma unidade de debug;

• Uso mais intenso da linguagem C;

• Uso de programas mais complexos, realizando benchmarks sobre o processador.

REFERÊNCIAS BIBLIOGRÁFICAS

ARORA, H.; KOTECHA, S.; SAMYAL, R. Dynamic branch prediction modeller for risc architecture. In: **2013 International Conference on Machine Intelligence and Research Advancement**. [S.l.]: IEEE, 2013. p. 397–401. ISBN 978-0-7695-5013-8.

BLEM, E.; MENON, J.; SANKARALINGAM, K. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In: **2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2013. p. 1–12. ISSN 1530-0897.

BROWN, S.; VRANESIC, Z. **Fundamentals of Digital Logic With VHDL Design**. 3. ed. [S.l.]: McGraw-Hill Education, 2008. 960 p. ISBN 9780073529530.

DENNIS, D. K. et al. Single cycle risc-v micro architecture processor and its fpga prototype. In: **2017 7th International Symposium on Embedded Computing and System Design (ISED)**. [S.l.]: IEEE, 2017. p. 1–5. ISBN 978-1-5386-3032-7. ISSN 2473-9413.

DIGILENT. **Nexys3 Board Reference Manual**. [S.l.], 2013. 22 p. Disponível em: <https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-3/nexys3_rm.pdf>.

ETHZURICH. **PULP - Parallel Ultra Low Power Platform**. 2015. Disponível em: <https://github.com/pulp-platform/pulpino>. Acesso em: 20 out. 2017.

GEORGE, A. D. An overview of riscv vs. cisc. In: **[1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory**. [S.l.: s.n.], 1990. p. 436–438. ISBN 0-8186-2038-2. ISSN 0094-2898.

HEXSEL, R. A. **Sistemas Digitais e Microprocessadores**. [S.l.]: UFPR, 2012. 306 p. ISBN 9788573353068.

JIMENEZ, D. A.; LIN, C. Dynamic branch prediction with perceptrons. In: **Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture**. [S.l.]: IEEE, 2001. p. 197–206. ISBN 0-7695-1019-1. ISSN 1530-0897.

KIAT, W. P. et al. A comprehensive analysis on data hazard for risc32 5-stage pipeline processor. In: **2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)**. [S.l.]: IEEE, 2017. p. 154–159. ISBN 978-1-5090-6231-7.

MATTHEWS, E.; SHANNON, L. Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features. In: **2017 27th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.]: IEEE, 2017. p. 1–4. ISBN 978-9-0903-0428-1. ISSN 1946-1488.

OLIVIERI, M. et al. Investigation on the optimal pipeline organization in risc-v multi-threaded soft processor cores. In: **2017 New Generation of CAS (NGCAS)**. [S.l.]: IEEE, 2017. p. 45–48. ISBN 978-1-5090-6447-2.

PANDEY, A. Study of data hazard and control hazard resolution techniques in a simulated five stage pipelined risc processor. In: **2016 International Conference on Inventive Computation Technologies (ICICT)**. [S.l.]: IEEE, 2016. v. 2, p. 1–4. ISBN 978-1-5090-1285-5.

PATTERSON, D. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In: **2018 IEEE International Solid - State Circuits Conference - (ISSCC)**. [S.l.: s.n.], 2018. p. 27–31. ISSN 2376-8606.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design - The Hardware Software/Interface**. 4. ed. [S.l.]: Morgan Kaufmann, 2004. 656 p. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 1558606041.

PEDRONI, V. A. **Circuit Design and Simulation with VHDL**. 2. ed. [S.l.]: MIT Press, 2010. 632 p. ISBN 9780262014335.

RAVEENDRAN, A. et al. A risc-v instruction set processor-micro-architecture design and analysis. In: **2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)**. [S.l.]: IEEE, 2016. p. 1–7. ISBN 978-1-5090-0033-3.

STALLINGS, W. **Computer Organization and Architecture - Designing for Performance**. 9. ed. [S.l.]: Pearson, 2012. 792 p. ISBN 9780132936330.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas Digitais : Princípios e Aplicações**. 11. ed. [S.l.]: Pearson / Prentice Hall, 2011. 840 p. ISBN 9788576059226.

VECTORBLOX. **ORCA: RISC-V**. 2015. Disponível em: <http://github.com/vectorblox/orca>. Acesso em: 20 out. 2017.

WATERMAN, A.; ASANOVIć, K. **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2**. [S.l.], 2017. 145 p. Disponível em: <https://github.com/riscv/riscv-isa-manual/blob/master/release/riscv-spec-v2.2.pdf>.

____. **The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.10**. [S.l.], 2017. 91 p. Disponível em: <https://github.com/riscv/riscv-isa-manual/blob/master/release/riscv-privileged-v1.10.pdf>.

XILINX. **HDL Coding Practices to Accelerate Design Performance**. [S.l.], 2006. 22 p. Disponível em: <https://www.xilinx.com/support/documentation/white_papers/wp231.pdf>.

____. **Timing Constraints User Guide**. [S.l.], 2009. 142 p. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug612.pdf>.

____. **Xilinx XST User Guide**. [S.l.], 2009. 485 p. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf>.

____. **Spartan-6 Family Overview**. [S.l.], 2011. 11 p. Disponível em: <https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf>.

____. **Spartan-6 FPGA Block RAM Resources - User Guide**. [S.l.], 2011. 34 p. Disponível em: <https://www.xilinx.com/support/documentation/user_guides/ug383.pdf>.

____. **Spartan-6 Libraries Guide for HDL Designs**. [S.l.], 2013. 317 p. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/spartan6_hdl.pdf>.

APÊNDICE A – PROGRAMAÇÃO

A.1 – CÓDIGO DE LINK

```
1  OUTPUT_ARCH( "riscv" )
2
3  /* Memory map
4    r = read, x = execute , w = write
5  */
6  MEMORY
7  {
8    ROM (rx) : ORIGIN = 0x00000000, LENGTH = 16K
9    RAM (xrw) : ORIGIN = 0x10010000, LENGTH = 16K
10 }
11
12 /* Stack and Heap sizes*/
13 stack_size = 2048;
14 heap_size = 512;
15
16 /* Stack beginning and ending */
17 _stack_start = ORIGIN(RAM)+LENGTH(RAM);
18 _stack_end = _stack_start - stack_size;
19
20 /* Output sections */
21 SECTIONS
22 {
23   /* text: test code section */
24   .text :
25   {
26     *(.boot)
27     *(.text)
28     *(.text*)
29     *(.rodata)
30     *(.rodata*)
31   } >ROM
32
33   /* data: Initialized data segment */
34   .data :
35   {
```

```
36      _sdata = .;
37      *(.data)
38      *(.data*)
39    } >RAM
40
41    /* bss: Uninitialized data segment */
42    .bss :
43    {
44      *(.bss)
45      *(.bss*)
46    } >RAM
47
48    /* heap: Heap segment */
49    .heap :
50    {
51      _heap_start = .;
52      . = . + heap_size;
53      _heap_end = .;
54    } >RAM
55  }
```

A.2 – CÓDIGO TESTE ASM - BASE E

```
1  .equ lui1, 0x00001
2  .equ lui2, 0x00011
3  .equ lui3,  0x00002
4  .equ lui4, 0x00003
5  .equ lui5, 0x00004
6  .equ lui6, 0x00005
7  .equ lui7, 0x00006
8  .equ lui8, 0x00007
9  .equ lui9, 0xFE080
10 .equ lui10, 0x00009
11 .equ lui11, 0xFE081
12 .equ lui12, 0x0000B
13 .equ lui13, 0x0000C
14 .equ lui14, 0x0000D
15 .equ lui15, 0xFFFFE
16 .equ auipc1, 0x00001
17 .equ auipc2, 0xFFFF7
```

```
18 .equ RAM_addr, 0x10010000
19 .equ UART, 0xFFFF0000
20 ## data section
21 .data
22 ## text section
23 .text
24    main:
25 ####################
26    reg_to_imm_test:
27 
28 ####################
29 ####################
30 ## addi instruction test
31    addi_test:
32    addi x0,x0,7
33    addi x1,x0,-7
34    addi x2,x0,14
35    addi x3,x0,-14
36    addi x4,x0,28
37    addi x5,x0,-28
38    addi x6,x0,56
39    addi x7,x0,-56
40    addi x8,x7,8
41    addi x9,x8,8
42    addi x10,x9,8
43    addi x11,x10,8
44    addi x12,x11,8
45    addi x13,x12,8
46    addi x14,x13,8 # x14 = 0
47    addi x15,x14,1 ## if (x15 = 1) -> ADDI -> OK!
48 
49 addi_uart:
50    add x13,x0,x15
51    jal x1,uart_write_loop
52 
53 ####################
54 ####################
55 ## stli instruction test
56 ## add instruction test
57 ## sltiu instructon test
58 ## sub instruction test
```

```
    addi x13,x0,-8
    addi x14,x0,0
    addi x15,x0,1

    stli_add_test:
    slti x0,x0,7 #0
    slti x1,x1,-7 #0
    slti x2,x2,14 #0
    slti x3,x3,-14 #0
    slti x4,x4,28 #0
    slti x5,x5,-28 #0
    slti x6,x6,56 #0
    slti x7,x7,-56 #0
    slti x8,x8,8 #1
    slti x9,x9,-8 #1
    slti x10,x10,8 #1
    slti x11,x11,-8 #1
    slti x12,x12,8 #1
    slti x13,x13,-8 #0
    slti x14,x14,8 #1
    slti x15,x15,-8 #0 # total de 6 regs. set

    add x1,x1,x0
    add x2,x2,x1
    add x3,x3,x2
    add x4,x4,x3
    add x5,x5,x4
    add x6,x6,x5
    add x7,x7,x6
    add x8,x8,x7
    add x9,x9,x8
    add x10,x10,x9
    add x11,x11,x10
    add x12,x12,x11
    add x13,x13,x12
    add x14,x14,x13
    add x15,x15,x14 # x15 = 6
    addi x15,x15,-5 # if (x15 = 1) -> SLTI , ADD -> OK!

stli_add_uart:
    add x13,x0,x15
```

```
100     jal x1,uart_write_loop
101
102     stliu_sub_test:
103     sltiu x0,x0,7 #0
104     sltiu x1,x1,-7 #1
105     sltiu x2,x2,14 #1
106     sltiu x3,x3,-14 #1
107     sltiu x4,x4,28 #1
108     sltiu x5,x5,-28 #1
109     sltiu x6,x6,56 #1
110     sltiu x7,x7,-56 #1
111     sltiu x8,x8,0 #0
112     sltiu x9,x9,-8 #1
113     sltiu x10,x10,8 #1
114     sltiu x11,x11,0 #0
115     sltiu x12,x12,8 #1
116     sltiu x13,x13,-8 #1
117     sltiu x14,x14,0 #0
118     sltiu x15,x15,-8 #1 # total de 12 regs. set
119
120     sub x1,x1,x0
121     sub x2,x2,x1
122     sub x3,x3,x2
123     sub x4,x4,x3
124     sub x5,x5,x4
125     sub x6,x6,x5
126     sub x7,x7,x6
127     sub x8,x8,x7
128     sub x9,x9,x8
129     sub x10,x10,x9
130     sub x11,x11,x10
131     sub x12,x12,x11
132     sub x13,x13,x12
133     sub x14,x14,x13
134     sub x15,x15,x14 # x15 = 2
135     addi x15,x15,-1 # if (x15 = 1) -> SLTIU , SUB -> OK!
136
137 sltiu_sub_uart:
138     add x13,x0,x15
139     jal x1,uart_write_loop
140
```

```
####################
####################
## auipc instruction test
   auipc_test:
   auipc x1,auipc1
   auipc x2,auipc2
   auipc x3,auipc1
   auipc x4,auipc2
   auipc x5,auipc1
   auipc x6,auipc2
   auipc x7,auipc1
   auipc x8,auipc2
   auipc x9,auipc1
   auipc x10,auipc2
   auipc x11,auipc1
   auipc x12,auipc2
   auipc x13,auipc1
   auipc x14,auipc2
   auipc x15,auipc1

   add x2,x2,x1
   add x3,x3,x2
   add x4,x4,x3
   add x5,x5,x4
   add x6,x6,x5
   add x7,x7,x6
   add x8,x8,x7
   add x9,x9,x8
   add x10,x10,x9
   add x11,x11,x10
   add x12,x12,x11
   add x13,x13,x12
   add x14,x14,x13
   add x15,x15,x14 # x15 = -219520
   li x14,219521
   add x15,x15,x14 # if (x15 = 1) -> AUPIC -> OK OBS: depende da posição no có
       digo (relativo a PC)

auipc_uart:
    add x13,x0,x15
    jal x1,uart_write_loop
```

```
181
182 ####################
183 ####################
184 ## ori instruction test
185 ## andi instruction test, dependente dos resultados de ori_test
186 ## xori instruction test
187     ori_test:
188     ori x1,x0,1
189     ori x2,x1,2
190     ori x3,x2,4
191     ori x4,x3,8
192     ori x5,x4,16
193     ori x6,x5,32
194     ori x7,x6,64 # x7 = 0xFFFF_007F = 127
195     ori x8,x0,-256
196     ori x9,x8,-128
197     ori x10,x9,-64
198     ori x11,x10,-32
199     ori x12,x11,-16
200     ori x13,x12,-8
201     ori x14,x13,-4
202     ori x15,x14,-2
203
204     sub x15,x7,x15 # x15 = 129
205     addi x15,x15,-128 # if (x15 = 1) -> ORI -> OK!
206
207 ori_uart:
208     add x13,x0,x15
209     jal x1,uart_write_loop
210
211     andi_test:
212     addi x1,x0,0x0000007F # x1 = 127
213     addi x8,x0,0xFFFFFFFE # x8 = -2
214
215     andi x2,x1,-4
216     andi x3,x2,-8
217     andi x4,x3,-16
218     andi x5,x4,-32
219     andi x6,x5,-64
220     andi x7,x6,-128 # x7 = 0x0000_0000
221     andi x8,x0,0x000007FE
```

```
222     andi x9,x8,0x000003FE
223     andi x10,x9,0x000001FE
224     andi x11,x10,0x000000FE
225     andi x12,x11,0x0000007E
226     andi x13,x12,0x0000003E
227     andi x14,x13,0x0000001E
228     andi x15,x14,0x00000008 # x15 = 0x0000_0000 = 0
229
230     add x15,x15,x7 #x15 = 0
231     addi x15,x15,1 # if (x15 = 1) -> ANDI -> OK!
232
233 andi_uart:
234     add x13,x0,x15
235     jal x1,uart_write_loop
236
237     xori_test:
238     xori x1,x0,1
239     xori x2,x1,3
240     xori x3,x2,6
241     xori x4,x3,12
242     xori x5,x4,24
243     xori x6,x5,48
244     xori x7,x6,96 # x7 = 64
245     xori x8,x0,0xFFFFF801
246     xori x9,x8,3
247     xori x10,x9,6
248     xori x11,x10,12
249     xori x12,x11,24
250     xori x13,x12,48
251     xori x14,x13,96
252     xori x15,x14,0xFFFFFFBF # x15 = 0x0000_07FF = 2047
253
254     sub x15,x15,x7 # x15 = 1983
255     addi x15,x15,-991
256     addi x15,x15,-991 # if (x15 = 1) -> XORI -> OK!
257
258 xori_uart:
259     add x13,x0,x15
260     jal x1,uart_write_loop
261
262 ####################
```

```
263 ####################
264 ## slli instruction test
265 ## srli instruction test
266 ## srla instruction test
267     slli_srli_test:
268     addi x1,x0,7
269     slli x2,x1,1
270     slli x3,x2,2
271     slli x4,x3,2
272     slli x5,x4,2
273     slli x6,x5,2
274     slli x7,x6,2
275     slli x8,x7,2
276     slli x9,x8,2
277     slli x10,x9,2
278     slli x11,x10,2
279     slli x12,x11,2
280     slli x13,x12,2
281     slli x14,x13,2 # 27:25 = "111"
282     slli x15,x14,2 # 29:27 = "111"
283 
284     srli x14,x15,2
285     srli x13,x14,2
286     srli x12,x13,2
287     srli x11,x12,2
288     srli x10,x11,2
289     srli x9,x10,2
290     srli x8,x9,2
291     srli x7,x8,2
292     srli x6,x7,2
293     srli x5,x6,2
294     srli x4,x5,2
295     srli x3,x4,1
296     srli x2,x3,1
297     srli x1,x2,1
298     srli x1,x1,2 # 2:0 = "111" = 7
299 
300     addi x1,x1,-6
301     add x15,x0,x1 # if (x15 = 1) -> SLLI , SRLI -> OK!
302 
303 slli_srli_uart:
```

```
    add x13,x0,x15
    jal x1,uart_write_loop

    srai_test:
    addi x1,x0,-256
    addi x8,x0,256

    srai x2,x1,1
    srai x3,x2,1
    srai x4,x3,1
    srai x5,x4,1
    srai x6,x5,1
    srai x7,x6,1 # x7 = -4
    srai x9,x8,1
    srai x10,x9,1
    srai x11,x10,1
    srai x12,x11,1
    srai x13,x12,1
    srai x14,x13,1
    srai x15,x14,1 # x15 = 2

    add x15,x15,x7 # x15 = -2
    addi x15,x15,3 # if (x15 = 1) -> SRAI -> OK

srai_uart:
    add x13,x0,x15
    jal x1,uart_write_loop

####################
####################
## lui instruction test
    lui_test:
    lui x1,lui1
    lui x2,lui2
    lui x3,lui3
    lui x4,lui4
    lui x5,lui5
    lui x6,lui6
    lui x7,lui7
    lui x8,lui8
    lui x9,lui9
```

```
345     lui x10,lui10
346     lui x11,lui11
347     lui x12,lui12
348     lui x13,lui13
349     lui x14,lui14
350     lui x15,lui15
351 
352     add x2,x2,x1
353     add x3,x3,x2
354     add x4,x4,x3
355     add x5,x5,x4
356     add x6,x6,x5
357     add x7,x7,x6
358     add x8,x8,x7
359     add x9,x9,x8
360     add x10,x10,x9
361     add x11,x11,x10
362     add x12,x12,x11
363     add x13,x13,x12
364     add x14,x14,x13
365     add x15,x15,x14 # x15 = -65695744
366     li x14,65695745
367     add x15,x15,x14 # if (x15 = 1) -> LUI -> OK!
368 
369 lui_uart:
370     add x13,x0,x15
371     jal x1,uart_write_loop
372 
373 ################################################################################
374 ################################################################################
375 ################################################################################
376 ################################################################################
377 ################################################################################
378     reg_to_reg_test:
379 
380 ####################
381 ####################
382 ## stl instruction test
383 ## stlu instruction test
384     stl_test:
385     li x1,-8
```

```
386     li x2,8
387     li x3,-8
388     li x4,8
389     li x5,-8
390     li x6,8
391     li x7,-8
392     li x8,8
393     li x9,-56
394     li x10,56
395     li x11,-28
396     li x12,28
397     li x13,-14
398     li x14,14
399     li x15,-7
400
401     slt x1,x1,x15 #1
402     slt x2,x2,x14 #1
403     slt x3,x3,x13 #0
404     slt x4,x4,x12 #1
405     slt x5,x5,x11 #0
406     slt x6,x6,x10 #1
407     slt x7,x7,x9 #0
408     slt x8,x8,x8 #0
409     slt x9,x9,x7 #1
410     slt x10,x10,x6 #0
411     slt x11,x11,x5 #1
412     slt x12,x12,x4 #0
413     slt x13,x13,x3 #1
414     slt x14,x14,x2 #0
415     slt x15,x15,x1 #1 # total de 8 regs. set
416
417     add x2,x2,x1
418     add x3,x3,x2
419     add x4,x4,x3
420     add x5,x5,x4
421     add x6,x6,x5
422     add x7,x7,x6
423     add x8,x8,x7
424     add x9,x9,x8
425     add x10,x10,x9
426     add x11,x11,x10
```

```
427     add x12,x12,x11
428     add x13,x13,x12
429     add x14,x14,x13
430     add x15,x15,x14 # sum = 8
431     addi x15,x15,-7 # if (x15 = 1) -> SLT -> OK!
432
433 slt_uart:
434     add x13,x0,x15
435     jal x1,uart_write_loop
436
437     stlu_test:
438     li x1,-8
439     li x2,0
440     li x3,-8
441     li x4,8
442     li x5,0
443     li x6,8
444     li x7,-8
445     li x8,0
446     li x9,5
447     li x10,56
448     li x11,-28
449     li x12,28
450     li x13,-14
451     li x14,14
452     li x15,-7
453
454     sltu x1,x1,x15 #1
455     sltu x2,x2,x14 #1
456     sltu x3,x3,x13 #0
457     sltu x4,x4,x12 #1
458     sltu x5,x5,x11 #1
459     sltu x6,x6,x10 #1
460     sltu x7,x7,x9 #0
461     sltu x8,x8,x8 #0
462     sltu x9,x9,x7 #0
463     sltu x10,x10,x6 #0
464     sltu x11,x11,x5 #0
465     sltu x12,x12,x4 #0
466     sltu x13,x13,x3 #0
467     sltu x14,x14,x2 #0
```

```
468     sltu x15,x15,x1 #0 # total de 5 regs. set
469
470     add x2,x2,x1
471     add x3,x3,x2
472     add x4,x4,x3
473     add x5,x5,x4
474     add x6,x6,x5
475     add x7,x7,x6
476     add x8,x8,x7
477     add x9,x9,x8
478     add x10,x10,x9
479     add x11,x11,x10
480     add x12,x12,x11
481     add x13,x13,x12
482     add x14,x14,x13
483     add x15,x15,x14 # sum = 5
484     addi x15,x15,-4 # if (x15 = 1) -> SLTU -> OK!
485
486 sltu_uart:
487     add x13,x0,x15
488     jal x1,uart_write_loop
489
490 ######################
491 ######################
492 ## or instruction test
493 ## and instruction test
494 ## xor instruction test
495     or_test:
496     li x1,1
497     li x2,2
498     li x3,4
499     li x4,8
500     li x5,16
501     li x6,32
502     li x7,64
503     li x8,-256
504     li x9,-128
505     li x10,-64
506     li x11,-32
507     li x12,-16
508     li x13,-8
```

```
509     li x14,-4
510     li x15,-2
511
512     or x1,x1,x0
513     or x2,x2,x1
514     or x3,x3,x2
515     or x4,x4,x3
516     or x5,x5,x4
517     or x6,x6,x5
518     or x7,x7,x6
519     or x8,x8,x0
520     or x9,x9,x8
521     or x10,x10,x9
522     or x11,x11,x10
523     or x12,x12,x11
524     or x13,x13,x12
525     or x14,x14,x13
526     or x15,x15,x14
527
528     sub x15,x7,x15 # x15 = 129
529     addi x15,x15,-128 # if (x15 = 1) -> OR -> OK!
530
531 or_uart:
532     add x13,x0,x15
533     jal x1,uart_write_loop
534
535     and_test:
536     li x1,0x0000007F # x1 = 127
537     li x8,0xFFFFFFFE # x8 = -2
538
539     li x2,-4
540     li x3,-8
541     li x4,-16
542     li x5,-32
543     li x6,-64
544     li x7,-128
545     li x9,0x000003FE
546     li x10,0x000001FE
547     li x11,0x000000FE
548     li x12,0x0000007E
549     li x13,0x0000003E
```

```
    li x14,0x0000001E
    li x15,0x00000008

    and x2,x2,x1
    and x3,x3,x2
    and x4,x4,x3
    and x5,x5,x4
    and x6,x6,x5
    and x7,x7,x6
    and x9,x9,x8
    and x10,x10,x9
    and x11,x11,x10
    and x12,x12,x11
    and x13,x13,x12
    and x14,x14,x13
    and x15,x15,x14 # x15 = 0x0000_0008 = 8

    add x15,x15,x7 #x15 = 8
    addi x15,x15,-7 # if (x15 = 1) -> AND -> OK!

and_uart:
    add x13,x0,x15
    jal x1,uart_write_loop

    xor_test:
    li x1,1
    li x2,3
    li x3,6
    li x4,12
    li x5,24
    li x6,48
    li x7,96
    li x8,0xFFFFF801
    li x9,3
    li x10,6
    li x11,12
    li x12,24
    li x13,48
    li x14,96
    li x15,0xFFFFFFBF

```

```
591     xor x1,x1,x0
592     xor x2,x2,x1
593     xor x3,x3,x2
594     xor x4,x4,x3
595     xor x5,x5,x4
596     xor x6,x6,x5
597     xor x7,x7,x6
598     xor x8,x8,x0
599     xor x9,x9,x8
600     xor x10,x10,x9
601     xor x11,x11,x10
602     xor x12,x12,x11
603     xor x13,x13,x12
604     xor x14,x14,x13
605     xor x15,x15,x14 # x15 = 0x0000_07FF = 2047
606 
607     sub x15,x15,x7 # x15 = 1983
608     addi x15,x15,-991
609     addi x15,x15,-991 # if (x15 = 1) -> XOR -> OK!
610 
611 xor_uart:
612     add x13,x0,x15
613     jal x1,uart_write_loop
614 
615 ####################
616 ####################
617 ## sll instruction test
618 ## srl instruction test
619 ## sra instruction test
620     sll_srl_test:
621     li x1,1
622     li x9,2
623 
624     sll x2,x1,x1
625     sll x3,x2,x9
626     sll x4,x3,x9
627     sll x5,x4,x9
628     sll x6,x5,x9
629     sll x7,x6,x9
630     sll x8,x7,x9
631     sll x10,x8,x9
```

```
632     sll x11,x10,x9
633     sll x12,x11,x9
634     sll x13,x12,x9
635     sll x14,x13,x9
636     sll x15,x14,x9 # x15(25) = '1'
637 
638     srl x14,x15,x9
639     srl x13,x14,x9
640     srl x12,x13,x9
641     srl x11,x12,x9
642     srl x10,x11,x9
643     srl x8,x10,x9
644     srl x7,x8,x9
645     srl x6,x7,x9
646     srl x5,x6,x9
647     srl x4,x5,x9
648     srl x3,x4,x1
649     srl x2,x3,x1
650     srl x1,x2,x1 # x1 = 4
651 
652     addi x15,x1,-3 # if (x15 = 1) -> SLL , SRL -> OK!
653 
654 sll_srl_uart:
655     add x13,x0,x15
656     jal x1,uart_write_loop
657 
658     sra_test:
659     addi x1,x0,-256
660     addi x8,x0,256
661     addi x15,x0,1
662 
663     sra x2,x1,x15
664     sra x3,x2,x15
665     sra x4,x3,x15
666     sra x5,x4,x15
667     sra x6,x5,x15
668     sra x7,x6,x15 # x7 = -4
669     addi x1,x0,1 ## troca shamt register de x15 para x1
670     sra x9,x8,x1
671     sra x10,x9,x1
672     sra x11,x10,x1
```

```
673     sra x12,x11,x1
674     sra x13,x12,x1
675     sra x14,x13,x1
676     sra x15,x14,x1 # x15 = 2
677 
678     add x15,x15,x7 # x15 = -2
679     addi x15,x15,3 # if (x15 = 1) -> SRA -> OK
680 
681 sra_uart:
682     add x13,x0,x15
683     jal x1,uart_write_loop
684 
685 ####################
686 ####################
687 ## beq instruction test
688 ## bne instruction test
689     addi x0,x0,7
690     addi x1,x0,-7
691     addi x2,x0,14
692     addi x3,x0,-14
693     addi x4,x0,28
694     addi x5,x0,-28
695     addi x6,x0,56
696     addi x7,x0,-56
697     addi x8,x0,8
698     addi x9,x0,7
699     addi x10,x0,6
700     addi x11,x0,5
701     addi x12,x0,5
702     addi x13,x0,5
703     addi x14,x0,4
704     addi x15,x0,4
705 
706     beq_test:
707     beq x2,x1,endbeq
708     beq x3,x2,endbeq
709     beq x4,x3,endbeq
710     beq x5,x4,endbeq
711     beq x6,x5,endbeq
712     beq x7,x6,endbeq
713     beq x8,x7,endbeq
```

```
714     beq x9,x8,endbeq
715     beq x10,x9,endbeq
716     beq x11,x10,endbeq
717     beq x12,x11,endbeq12
718     addi x15,x15,1
719 endbeq12:
720     beq x13,x12,endbeq13
721     addi x15,x15,1
722 endbeq13:
723     beq x14,x13,endbeq
724     beq x15,x14,endbeq15
725 endbeq:
726     addi x15,x15,-15
727 endbeq15:
728     addi x15,x15,-3 # if (x15 = 1) -> BEQ -> OK!
729 
730 beq_uart:
731     add x13,x0,x15
732     jal x1,uart_write_loop
733 
734     bne_test:
735     addi x0,x0,7
736     addi x1,x0,7
737     addi x2,x0,7
738     addi x3,x0,7
739     addi x4,x0,7
740     addi x5,x0,7
741     addi x6,x0,7
742     addi x7,x0,7
743     addi x8,x0,7
744     addi x9,x0,7
745     addi x10,x0,7
746     addi x11,x0,7
747     addi x12,x0,5
748     addi x13,x0,4
749     addi x14,x0,4
750     addi x15,x0,3
751     bne x2,x1,endbne
752     bne x3,x2,endbne
753     bne x4,x3,endbne
754     bne x5,x4,endbne
```

```
755     bne x6,x5,endbne
756     bne x7,x6,endbne
757     bne x8,x7,endbne
758     bne x9,x8,endbne
759     bne x10,x9,endbne
760     bne x11,x10,endbne
761     bne x12,x11,endbne12
762     addi x15,x15,1
763 endbne12:
764     bne x13,x12,endbne13
765     addi x15,x15,1
766 endbne13:
767     bne x14,x13,endbne
768     bne x15,x14,endbne15
769 endbne:
770     addi x15,x15,-15
771 endbne15:
772     addi x15,x15,-2 # if (x15 = 1) -> BNE -> OK!
773 
774 bne_uart:
775     add x13,x0,x15
776     jal x1,uart_write_loop
777 
778 ####################
779 ####################
780 ## blt instruction test
781 ## bltu insutrction test
782     addi x0,x0,7
783     addi x1,x0,8
784     addi x2,x0,9
785     addi x3,x0,10
786     addi x4,x0,11
787     addi x5,x0,12
788     addi x6,x0,13
789     addi x7,x0,14
790     addi x8,x0,15
791     addi x9,x0,16
792     addi x10,x0,17
793     addi x11,x0,18
794     addi x12,x0,6
795     addi x13,x0,-55
```

```
796     addi x14,x0,5
797     addi x15,x0,4
798
799     blt_test:
800     blt x2,x1,endblt # branch if x2 < x1 ---> blt/bltu/bge/bgeu rs2,rs1,offset
            if rs1 < rs2
801     blt x3,x2,endblt
802     blt x4,x3,endblt
803     blt x5,x4,endblt
804     blt x6,x5,endblt
805     blt x7,x6,endblt
806     blt x8,x7,endblt
807     blt x9,x8,endblt
808     blt x10,x9,endblt
809     blt x11,x10,endblt
810     blt x12,x11,endblt12
811     addi x15,x15,1
812 endblt12:
813     blt x13,x12,endblt13
814     addi x15,x15,1
815 endblt13:
816     blt x14,x13,endblt
817     blt x15,x14,endblt15
818 endblt:
819     addi x15,x15,-15
820 endblt15:
821     addi x15,x15,-3 # if (x15 = 1) -> BLT -> OK!
822
823 blt_uart:
824     add x13,x0,x15
825     jal x1,uart_write_loop
826
827     bltu_test:
828     addi x1,x0,8
829     addi x13,x0,-55
830     addi x14,x0,5
831     addi x15,x0,4 #x15 = 4
832     bltu x2,x1,endbltu
833     bltu x3,x2,endbltu
834     bltu x4,x3,endbltu
835     bltu x5,x4,endbltu
```

```
836     bltu x6,x5,endbltu
837     bltu x7,x6,endbltu
838     bltu x8,x7,endbltu
839     bltu x9,x8,endbltu
840     bltu x10,x9,endbltu
841     bltu x11,x10,endbltu
842     bltu x12,x11,endbltu12
843     addi x15,x15,1
844 endbltu12:
845     bltu x13,x12,endbltu
846     bltu x14,x13,endbltu14
847     addi x15,x15,-7
848 endbltu14:
849     bltu x15,x14,endbltu15
850 endbltu:
851     addi x15,x15,-15
852 endbltu15:
853     addi x15,x15,-3 # if (x15 = 1) -> BLTU -> OK!
854 
855 bltu_uart:
856     add x13,x0,x15
857     jal x1,uart_write_loop
858 
859 ####################
860 ####################
861 ## blge instruction test
862 ## blegu instruction test
863     bge_test:
864     addi x1,x0,17
865     addi x2,x0,16
866     addi x3,x0,15
867     addi x4,x0,14
868     addi x5,x0,13
869     addi x6,x0,12
870     addi x7,x0,11
871     addi x8,x0,10
872     addi x9,x0,9
873     addi x10,x0,8
874     addi x11,x0,7
875     addi x12,x0,7
876     addi x13,x0,11
```

```
877     addi x14,x0,-55
878     addi x15,x0,44
879 
880     bge x2,x1,endbge
881     bge x3,x2,endbge
882     bge x4,x3,endbge
883     bge x5,x4,endbge
884     bge x6,x5,endbge
885     bge x7,x6,endbge
886     bge x8,x7,endbge
887     bge x9,x8,endbge
888     bge x10,x9,endbge
889     bge x11,x10,endbge
890     bge x12,x11,endbge12
891     addi x15,x15,1
892 endbge12:
893     bge x13,x12,endbge13
894     addi x15,x15,1
895 endbge13:
896     bge x14,x13,endbge
897     bge x15,x14,endbge15
898 endbge:
899     addi x15,x15,-15
900 endbge15:
901     addi x15,x15,-43 # if (x15 = 1) -> BGE -> OK!
902 
903 bge_uart:
904     add x13,x0,x15
905     jal x1,uart_write_loop
906 
907     bgeu_test:
908     addi x1,x0,17
909     addi x13,x0,11
910     addi x14,x0,-55
911     addi x15,x0,4
912     bgeu x2,x1,endbgeu
913     bgeu x3,x2,endbgeu
914     bgeu x4,x3,endbgeu
915     bgeu x5,x4,endbgeu
916     bgeu x6,x5,endbgeu
917     bgeu x7,x6,endbgeu
```

```
    bgeu x8,x7,endbgeu
    bgeu x9,x8,endbgeu
    bgeu x10,x9,endbgeu
    bgeu x11,x10,endbgeu
    bgeu x12,x11,endbgeu12
    addi x15,x15,1
endbgeu12:
    bgeu x13,x12,endbgeu13
    addi x15,x15,1
endbgeu13:
    bgeu x14,x13,endbgeu14
    addi x15,x15,1
endbgeu14:
    bgeu x15,x14,endbgeu15
endbgeu:
    addi x15,x15,-15
endbgeu15:
    addi x15,x15,12 # if (x15 = 1) -> BGEU -> OK!

bgeu_uart:
    add x13,x0,x15
    jal x1,uart_write_loop

####################
####################
## jal instruction test
## jalr instruction test
    jal_test:
    jal x1,1f
1: jal x2,2f
2: jal x3,3f
3: jal x4,4f
4: jal x5,5f
5: jal x6,6f
6: jal x7,7f
7: jal x8,8f
8: jal x9,9f
9: jal x10,10f
10: jal x11,11f
11: jal x12,12f
12: jal x13,13f
```

```
13: jal x14,14f
14:
    sub x2,x2,x1
    sub x4,x4,x3
    sub x6,x6,x5
    sub x8,x8,x7
    sub x10,x10,x9
    sub x12,x12,x11
    sub x14,x14,x13

    add x4,x4,x2
    add x6,x6,x4
    add x8,x8,x6
    add x10,x10,x8
    add x12,x12,x10
    add x14,x14,x12
    addi x14,x14,-27 ## if (x15 = 1) -> JAL -> OK!

jal_uart:
    add x13,x0,x14
    jal x1,uart_write_loop

    jal_r_test:
   li x15,10
   jal x1,0f # x1 = 2
0:
   jalr x0,x1,8 # x1 = 8
   addi x15,x0,5 # deve pular todos os addi
   jalr x2,x1,20
   addi x15,x15,5
   addi x15,x15,5
   jalr x3,x1,36 # 20
   addi x15,x15,5
   addi x15,x15,5
   addi x15,x15,5
   jalr x4,x1,56 # 36
   addi x15,x15,5
   addi x15,x15,5
   addi x15,x15,5
   addi x15,x15,5
   jalr x5,x1,80 # 56
```

```
1000    addi x15,x15,3
1001    addi x15,x15,3
1002    addi x15,x15,3
1003    addi x15,x15,3
1004    addi x15,x15,3
1005    jalr x6,x1,100 # 80
1006    addi x15,x15,3
1007    addi x15,x15,3
1008    addi x15,x15,3
1009    addi x15,x15,3
1010    jalr x7,x1,116 # 100
1011    addi x15,x15,3
1012    addi x15,x15,3
1013    addi x15,x15,3
1014    jalr x8,x1,128 # 116
1015    addi x15,x15,3
1016    addi x15,x15,3
1017    jalr x9,x1,136 # 128
1018    addi x15,x15,3
1019    jalr x10,x1,144 # 136
1020    addi x15,x15,3
1021    jalr x11,x1,152 # 144
1022    addi x15,x15,3
1023    jalr x12,x1,160 # 152
1024    addi x15,x15,3
1025    jalr x13,x1,168 # 160
1026    addi x15,x15,3
1027    jalr x14,x1,176 # 168
1028    addi x15,x15,3
1029
1030    sub x2,x2,x1 # x2 = 8
1031    sub x4,x4,x3 # x4 = 16
1032    sub x6,x6,x5 # x6 = 24
1033    sub x8,x8,x7 # x8 = 16
1034    sub x10,x10,x9 # x10 = 8
1035    sub x12,x12,x11 # x12 = 8
1036    sub x14,x14,x13 # x14 = 8
1037
1038    add x4,x4,x2
1039    add x6,x6,x4
1040    add x8,x8,x6
```

```
   add x10,x10,x8
   add x12,x12,x10
   add x14,x14,x12 # x14 = 92
   addi x14,x14,-91 ## if (x15 = 1) -> JALR -> OK!

jalr_uart:
   add x13,x0,x14
   jal x1,uart_write_loop

####################
####################
    li:
    li x1,0x12345678
    li x2,0x23456789
    li x3,0x3456789a
    li x4,0x456789ab
    li x5,0x56789abc
    li x6,0x6789abcd
    li x7,0x789abcde
    li x8,0x89abcdef
    li x9,0x9abcdef0
    li x10,0xabcdef01
    li x11,0xbcdef012
    li x12,0xcdef0123
    li x13,0xdef01234
    li x14,0xef012345
    li x15,RAM_addr

    sw_lw_test:
    sw x1,0(x15)
    sw x2,4(x15)
    sw x3,8(x15)
    sw x4,12(x15)
    sw x5,16(x15)
    sw x6,20(x15)
    sw x7,24(x15)
    sw x8,28(x15)
    sw x9,32(x15)
    sw x10,36(x15)
    sw x11,40(x15)
    sw x12,44(x15)
```

```
1082     sw x13,48(x15)
1083     sw x14,52(x15)
1084
1085     lw x14,0(x15)
1086     lw x13,4(x15)
1087     lw x12,8(x15)
1088     lw x11,12(x15)
1089     lw x10,16(x15)
1090     lw x9,20(x15)
1091     lw x8,24(x15)
1092     lw x7,28(x15)
1093     lw x6,32(x15)
1094     lw x5,36(x15)
1095     lw x4,40(x15)
1096     lw x3,44(x15)
1097     lw x2,48(x15)
1098     lw x1,52(x15)
1099
1100     add x2,x2,x1
1101     add x3,x3,x2
1102     add x4,x4,x3
1103     add x5,x5,x4
1104     add x6,x6,x5
1105     add x7,x7,x6
1106     add x8,x8,x7
1107     add x9,x9,x8
1108     add x10,x10,x9
1109     add x11,x11,x10
1110     add x12,x12,x11
1111     add x13,x13,x12
1112     add x14,x14,x13
1113     li x13,248153658
1114     sub x14,x14,x13 # if (x15 = 1) -> SW , LW -> OK!
1115
1116 sw_lw_uart:
1117     add x13,x0,x14
1118     jal x1,uart_write_loop
1119
1120     lb_lbu_test:
1121     li x15,RAM_addr
1122
```

```
    lb x14,0(x15)
    lbu x13,4(x15)
    lb x12,8(x15)
    lbu x11,12(x15)
    lb x10,16(x15)
    lbu x9,20(x15)
    lb x8,24(x15)
    lbu x7,28(x15)
    lb x6,32(x15)
    lbu x5,36(x15)
    lb x4,40(x15)
    lbu x3,44(x15)
    lb x2,48(x15)
    lbu x1,52(x15)

    add x2,x2,x1
    add x3,x3,x2
    add x4,x4,x3
    add x5,x5,x4
    add x6,x6,x5
    add x7,x7,x6
    add x8,x8,x7
    add x9,x9,x8
    add x10,x10,x9
    add x11,x11,x10
    add x12,x12,x11
    add x13,x13,x12
    add x14,x14,x13
    addi x14,x14,-826 # if (x15 = 1) -> LB , LBU -> OK!

lb_lbu_uart:
    add x13,x0,x14
    jal x1,uart_write_loop

    li_2:
    li x1,0x12345678
    li x2,0x23456789
    li x3,0x3456789a
    li x4,0x456789ab
    li x5,0x56789abc
    li x6,0x6789abcd
```

```
1164    li x7,0x789abcde
1165    li x8,0x89abcdef
1166    li x9,0x9abcdef0
1167    li x10,0xabcdef01
1168    li x11,0xbcdef012
1169    li x12,0xcdef0123
1170    li x13,0xdef01234
1171    li x14,0xef012345
1172    li x15,RAM_addr
1173
1174    sb_sh_lh_lhu_test:
1175    sb x1,56(x15)
1176    sb x2,60(x15)
1177    sb x3,64(x15)
1178    sb x4,68(x15)
1179    sh x5,72(x15)
1180    sh x6,76(x15)
1181    sh x7,80(x15)
1182    sh x8,84(x15)
1183    sb x9,88(x15)
1184    sb x10,92(x15)
1185    sb x11,96(x15)
1186    sh x12,100(x15)
1187    sh x13,104(x15)
1188    sh x14,108(x15)
1189
1190    lh x14,56(x15)
1191    lh x13,60(x15)
1192    lhu x12,64(x15)
1193    lhu x11,68(x15)
1194    lhu x10,72(x15)
1195    lhu x9,76(x15)
1196    lh x8,80(x15)
1197    lh x7,84(x15)
1198    lh x6,88(x15)
1199    lh x5,92(x15)
1200    lhu x4,96(x15)
1201    lhu x3,100(x15)
1202    lh x2,104(x15)
1203    lh x1,108(x15)
1204
```

```
1205     add x2,x2,x1
1206     add x3,x3,x2
1207     add x4,x4,x3
1208     add x5,x5,x4
1209     add x6,x6,x5
1210     add x7,x7,x6
1211     add x8,x8,x7
1212     add x9,x9,x8
1213     add x10,x10,x9
1214     add x11,x11,x10
1215     add x12,x12,x11
1216     add x13,x13,x12
1217     add x14,x14,x13
1218     li x13,-68410
1219     add x14,x14,x13
1220
1221     sb_sh_lh_lhu_uart:
1222     add x13,x0,x14
1223     jal x1,uart_write_loop
1224
1225 ########################################################
1226 ###### inf loop
1227     inf_loop:
1228     jal x1,inf_loop
1229
1230 ################################
1231 #### Leitura de UART_RxT, dados recebidos pela UART
1232     uart_write_loop:
1233     li x15,UART
1234 loop2_b:
1235     lw x14,0(x15)
1236     andi x14,x14,8 # bit 1 indica que está pronta para enviar, quando = '1'
1237     addi x14,x14,-8
1238     bne x14,x0,loop2_b
1239     addi x13,x13,30
1240     addi x13,x13,18 # ascii table offset
1241     sw x13,4(x15) # x13 = dado escrito
1242     jalr x0,x1,0
```

```
.equ lui1, 0x00001
.equ lui3,  0x00002
.equ lui4, 0x00003
.equ lui5, 0x00004
.equ lui6, 0x00005
.equ lui7, 0x00006
.equ lui8, 0x00007
.equ lui9, 0x00008
.equ lui10, 0x00009
.equ lui11, 0x0000A
.equ lui12, 0x0000B
.equ lui13, 0x0000C
.equ lui14, 0x0000D
.equ lui15, 0xFFFFE
.equ UART_HI, 0xFFFF0
.equ UART_RxT_LO, 0x004
## data section
.data
## text section
.text
   main:
####################
## c.addi  instruction test, x0 - x15
## c.add  instruction test, x0 - x15
## c.mv  instruction test, x0 - x15
## c.j   instruction test

   c_addi_test:
   c.addi x1,-7
   c.addi x2,14
   c.addi x3,-14
   c.addi x4,28
   c.addi x5,-28
   c.addi x6,30
   c.addi x7,-30
   c.addi x8,1
   c.addi x9,2
   c.addi x10,3
   c.addi x11,4
   c.addi x12,5
   c.addi x13,6
```

```
42      c.addi x14,8
43      c.addi x15,1
44
45      c_add_test:
46      c.add x2,x1
47      c.add x3,x2
48      c.add x4,x3
49      c.add x5,x4
50      c.add x6,x5
51      c.add x7,x6
52      c.add x8,x7
53      c.add x9,x8
54      c.add x10,x9
55      c.add x11,x10
56      c.add x12,x11
57      c.add x13,x12
58      c.add x14,x13
59      c.add x15,x14
60      c.addi x15,-22 # if (x15 = 1) -> c.addi , c.add -> OK!
61
62  addi_add_uart:
63      c.mv x13,x15
64      c.jal uart_write_loop
65
66      c.li x15,1
67      c.nop
68      c.nop
69      c.nop
70      c.nop # if (x15 = 1) -> c.nop -> OK!
71
72  nop_uart:
73      c.mv x13,x15
74      c.jal uart_write_loop
75
76      c_mv_test:
77      c.li x1,-7
78      c.mv x2,x1
79      c.mv x3,x2
80      c.mv x4,x3
81      c.mv x5,x4
82      c.mv x6,x5
```

```
83      c.mv x7,x6
84      c.mv x8,x7
85      c.mv x9,x8
86      c.mv x10,x9
87      c.mv x11,x10
88      c.mv x12,x11
89      c.mv x13,x12
90      c.mv x14,x13
91      c.mv x15,x14
92      c.addi x15,8 # if (x15 = 1) -> c.mv -> OK!
93
94  mv_uart:
95      c.mv x13,x15
96      c.jal uart_write_loop
97
98      c.li x15,1
99      c_j_test0:
100     c.j c_j_test1
101     c.addi x15,-15 # não deve executar
102     c_j_test2:
103     c.addi x15,2
104     c.j c_j_test3
105     c_j_test1:
106     c.j c_j_test2
107     c_j_test3:
108     c.addi x15,-2 # if (x15 = 1) -> c.j -> OK!
109
110 j_uart:
111     c.mv x13,x15
112     c.jal uart_write_loop
113
114 ####################
115 ####################
116 ## c.addi16sp  instruction test
117 ## c.addi4spn  instruction test, x8 - x15
118 ## c.sub   instruction test, x8 - x15
119 ## c.jal   instruction test
120     c_addi16sp_test:
121     c.li x2,1 # x2 = 1
122     c.addi16sp x2,-16
123     c.addi16sp x2,32
```

```
124     c.addi x2,-16
125     c.mv x15,x2 # if (x15 = 1) -> c.addi16sp -> OK!
126 
127 addi16sp_uart:
128     c.mv x13,x15
129     c.jal uart_write_loop
130 
131     c_addi4spn_sub_test:
132     # x2 = 1
133     c.addi4spn x8,x2,4
134     c.addi4spn x9,x2,8
135     c.addi4spn x10,x2,12
136     c.addi4spn x11,x2,16
137     c.addi4spn x12,x2,32
138     c.addi4spn x13,x2,64
139     c.addi4spn x14,x2,64
140     c.addi4spn x15,x2,16
141 
142     c.sub x9,x8
143     c.sub x10,x9
144     c.sub x11,x10
145     c.sub x12,x11
146     c.sub x13,x12
147     c.sub x14,x13
148     c.sub x15,x14 # x15 = -8
149     c.addi x15,9 # if (x15 = 1) -> c.addi4spn e c.sub -> OK!
150 
151 addi4spn_sub_uart:
152     c.mv x13,x15
153     c.jal uart_write_loop
154 
155   c_jal_test0:
156     c.jal c_jal_test1 # x1 = PC+2
157     c_jal_test2:
158     c.mv x14,x1
159     c.sub x14,x15 # x14 = (PC+14) - (PC+2) = 12
160     c.jal c_jal_test3
161     c_jal_test1:
162     c.mv x15,x1 # x15 = PC+2
163     c.jal c_jal_test2 # x1 = PC+14
164     c_jal_test3:
```

```
165     c.addi x14,-9
166     c.mv x15,x14 # if (x15 = 1) -> c.jal -> OK!
167
168 jal_uart:
169     c.mv x13,x15
170     c.jal uart_write_loop
171
172 ####################
173 ####################
174 ## x) c.li instruction test, x0 - x15
175     c_li_test:
176     c.li x1,-7
177     c.li x2,14
178     c.li x3,-14
179     c.li x4,28
180     c.li x5,-28
181     c.li x6,21
182     c.li x7,-13
183     c.li x8,8
184     c.li x9,8
185     c.li x10,8
186     c.li x11,8
187     c.li x12,8
188     c.li x13,8
189     c.li x14,8
190     c.li x15,1
191
192     c.add x2,x1
193     c.add x3,x2
194     c.add x4,x3
195     c.add x5,x4
196     c.add x6,x5
197     c.add x7,x6 # if (x7 = 1) -> c.li PARA X0 <-> X7
198
199     c.mv x8,x7  # x8 = 1
200     c.sub x8,x15
201     c.sub x9,x8
202     c.sub x10,x9
203     c.sub x11,x10
204     c.sub x12,x11
205     c.sub x13,x12
```

```
    c.sub x14,x13
    c.sub x15,x14 # if (x15 = 1) -> c.li PARA X8 <-> X15
    c.add x15,x7
    c.addi x15,-1 # if (x15 = 1) -> c.li -> OK!

li_uart:
    c.mv x13,x15
    c.jal uart_write_loop

####################
####################
## c.beqz  instruction test, x8 - x15
## c.bnez  instruction test, x8 - x15

    c_beqz_test:
    c.li x7,0
    c.li x8,-3
    c.li x9,-2
    c.li x10,-1
    c.li x11,0
    c.li x12,0
    c.li x13,0
    c.li x14,1
    c.li x15,2

    c.beqz x8,c_beqz_test1
    c.addi x7,1
    c.beqz x9,c_beqz_test1
    c.addi x7,2
    c.beqz x10,c_beqz_test1
    c.addi x7,3
    c.beqz x11,c_beqz_test1
    c.addi x7,4
    c_beqz_test1:
    c.beqz x12,c_beqz_test2
    c.addi x7,5
    c_beqz_test2:
    c.beqz x13,c_beqz_test3
    c.addi x7,6
    c_beqz_test3:
    c.beqz x14,c_beqz_test4
```

```
247     c.addi x7,7
248     c.beqz x15,c_beqz_test4
249     c.addi x7,8
250     c_beqz_test4:
251     c.addi x7,-20
252     c.mv x15,x7 # if (x15 = 1) -> c.beqz -> OK!
253
254 beqz_uart:
255     c.mv x13,x15
256     c.jal uart_write_loop
257
258     c_bnez_test:
259     c.li x7,0
260     c.li x8,0
261     c.li x9,0
262     c.li x10,0
263     c.li x11,-15
264     c.li x12,-2
265     c.li x13,19
266     c.li x14,0
267     c.li x15,0
268
269     c.bnez x8,c_bnez_test1
270     c.addi x7,1
271     c.bnez x9,c_bnez_test1
272     c.addi x7,2
273     c.bnez x10,c_bnez_test1
274     c.addi x7,3
275     c.bnez x11,c_bnez_test1
276     c.addi x7,4
277     c_bnez_test1:
278     c.bnez x12,c_bnez_test2
279     c.addi x7,5
280     c_bnez_test2:
281     c.bnez x13,c_bnez_test3
282     c.addi x7,6
283     c_bnez_test3:
284     c.bnez x14,c_bnez_test4
285     c.addi x7,7
286     c.bnez x15,c_bnez_test4
287     c.addi x7,8
```

```
288     c_bnez_test4:
289     c.addi x7,-20
290     c.mv x15,x7 # if (x15 = 1) -> c.bnez -> OK!
291 
292 bnez_uart:
293     c.mv x13,x15
294     c.jal uart_write_loop
295 
296 ####################
297 ####################
298 ## x) c.or instruction test, x8 - x15
299 ## x) c.and instruction test, x8 - x15
300 ## x) c.xor instruction test, x8 - x15
301 ## x) c.andi instruction test, x8 - x15
302 
303   c.li x8,1
304     c.li x9,4
305     c.li x10,2
306     c.li x11,8
307     c.li x12,1
308     c.li x13,1
309     c.li x14,0
310     c.li x15,7
311 
312     c_or_test:
313     c.or x9,x8
314     c.or x10,x9
315     c.or x11,x10
316     c.or x12,x11
317     c.or x13,x12
318     c.or x14,x13
319     c.or x15,x14
320     c.addi x15,-14 # if (x15 = 1) -> c.or -> OK!
321 
322 or_uart:
323     c.mv x7,x13
324     c.mv x13,x15
325     c.jal uart_write_loop
326     c.mv x13,x7
327 
328   c.li x8,15
```

```
329     c.li x15,4
330
331     c_and_test:
332     c.and x9,x8
333     c.and x10,x9
334     c.and x11,x10
335     c.and x12,x11
336     c.and x13,x12
337     c.and x14,x13
338     c.and x15,x14 # x15 = 0
339     c.addi x15,1 # if (x15 = 1) -> c.and -> OK!
340
341 and_uart:
342     c.mv x7,x13
343     c.mv x13,x15
344     c.jal uart_write_loop
345     c.mv x13,x7
346
347   c.li  x14,24
348     c.li x15,0
349
350     c_xor_test:
351     c.xor x8,x8
352     c.xor x9,x8
353     c.xor x10,x9
354     c.xor x11,x10
355     c.xor x12,x11
356     c.xor x13,x12
357     c.xor x14,x13
358     c.xor x15,x14
359     c.addi x15,-28 # if (x15 = 1) -> c.xor -> OK!
360
361 xor_uart:
362     c.mv x13,x15
363     c.jal uart_write_loop
364
365     andi_test:
366     c.li x8,-1
367     c.li x9,-1
368     c.li x10,-1
369     c.li x11,-1
```

```
370     c.li x12,-1
371     c.li x13,-1
372     c.li x14,-1
373     c.li x15,-1 # 0xFFFFFFFF
374 
375     c.andi x8,0x00000001
376     c.andi x9,0x00000002
377     c.andi x10,0x00000004
378     c.andi x11,0x00000008
379     c.andi x12,-1 # 0xFFFFFFFF
380     c.andi x13,-2 # 0xFFFFFFFE
381     c.andi x14,-4 # 0xFFFFFFFC
382     c.andi x15,-8 # 0xFFFFFFF8
383 
384     c.addi x8,1
385     c.sub x9,x8
386     c.sub x10,x9
387     c.sub x11,x10
388     c.sub x12,x11
389     c.sub x13,x12
390     c.sub x14,x13
391     c.sub x15,x14 # x15 = -1
392     c.addi x15,2 # if (x15 = 1) -> c.andi -> OK!
393 
394 andi_uart:
395     c.mv x13,x15
396     c.jal uart_write_loop
397 
398 ####################
399 ####################
400 ## x) c.slli instruction test, x0 - x15
401 ## x) c.srli instruction test, x8 - x15
402 ## x) srla instruction test, x8 - x15
403     c_slli_test:
404     c.li x1,3
405     c.li x2,3
406     c.li x3,3
407     c.li x4,3
408     c.li x5,3
409     c.li x6,3
410     c.li x7,3
```

```
411     c.li x8,4
412     c.li x9,4
413     c.li x10,4
414     c.li x11,4
415     c.li x12,4
416     c.li x13,4
417     c.li x14,4
418     c.li x15,4
419
420     c.slli x1,1
421     c.slli x2,2
422     c.slli x3,3
423     c.slli x4,4
424     c.slli x5,5
425     c.slli x6,6
426     c.slli x7,7
427     c.slli x8,1
428     c.slli x9,1
429     c.slli x10,2
430     c.slli x11,3
431     c.slli x12,4
432     c.slli x13,5
433     c.slli x14,6
434     c.slli x15,7
435
436     c.add x2,x1
437     c.add x3,x2
438     c.add x4,x3
439     c.add x5,x4
440     c.add x6,x5
441     c.add x7,x6 # x7 = 762
442     c.add x9,x8
443     c.add x10,x9
444     c.add x11,x10
445     c.add x12,x11
446     c.add x13,x12
447     c.add x14,x13
448     c.add x15,x14 # x15 = 1024
449     c.mv x14,x7
450     c.sub x15,x14 # x15 = 262
451     c.addi x13,5
```

```
452     c.sub x15,x13 # if x = 1, OK slli
453
454 slli_uart:
455     c.mv x13,x15
456     jal uart_write_loop
457
458     c_srli_test:
459     c.li x8,-16 #
460     c.li x9,-15 #
461     c.li x10,14 #
462     c.li x11,13 #
463     c.li x12,1
464     c.li x13,2
465     c.li x14,4
466     c.li x15,7
467
468     c.srli x8,30 # 0x0000_0003 = 3
469     c.srli x9,29 # 0x0000_0007 = 7
470     c.srli x10,28 # 0x0000_0000 = 0
471     c.srli x11,27 # 0x0000_0000 = 0
472     c.srli x12,8 # 0x0000_0000 = 0
473     c.srli x13,3 # 0x0000_0000 = 0
474     c.srli x14,2 # 0x0000_0001 = 1
475     c.srli x15,1 # 0x0000_0003 = 3
476
477     c.add x9,x8
478     c.add x10,x9
479     c.add x11,x10
480     c.add x12,x11
481     c.add x13,x12
482     c.add x14,x13
483     c.add x15,x14 # x15 = 14
484     c.addi x15,-13 # if (x15 = 1) -> c.srli -> OK!
485
486 srli_uart:
487     c.mv x13,x15
488     c.jal uart_write_loop
489
490     c_srai_test:
491     c.li x8,-16 # 0xFFFFFFF0
492     c.li x9,15 # 0xFFFFFFF1
```

```
493     c.li x10,14 #
494     c.li x11,13 #
495     c.li x12,1
496     c.li x13,2
497     c.li x14,4
498     c.li x15,8
499
500     c.srai x8,5 # -1
501     c.srai x9,4 # 0
502     c.srai x10,2 # 3
503     c.srai x11,1 # 6
504     c.srai x12,5 # 0
505     c.srai x13,4 # 0
506     c.srai x14,2 # 1
507     c.srai x15,1 # 4
508
509     c.add x9,x8
510     c.add x10,x9
511     c.add x11,x10
512     c.add x12,x11
513     c.add x13,x12
514     c.add x14,x13
515     c.add x15,x14
516     c.addi x15,-12 # if (x15 = 1) -> c.srai -> OK!
517
518 srai_uart:
519     c.mv x13,x15
520     c.jal uart_write_loop
521
522 ####################
523 ####################
524 ## c.lui  instruction text
525   c_lui_test:
526   c.lui  x1,lui1
527   c.lui  x3,lui3
528   c.lui  x4,lui4
529   c.lui  x5,lui5
530   c.lui  x6,lui6
531   c.lui  x7,lui7
532   c.lui  x8,lui8
533   c.lui  x9,lui9
```

```
  c.lui  x10,lui10
  c.lui  x11,lui11
  c.lui  x12,lui12
  c.lui  x13,lui13
  c.lui  x14,lui14
  c.lui  x15,lui15

    c.add x3,x1
    c.add x4,x3
    c.add x5,x4
    c.add x6,x5
    c.add x7,x6
    c.add x8,x7
    c.add x9,x8
    c.add x10,x9
    c.add x11,x10
    c.add x12,x11
    c.add x13,x12
    c.add x14,x13

  c.lui  x13,0xFFFF2
  c.xor x15,x13
    c.add x15,x14
  c.srli x15,16
    c.addi x15,-5 # if (x15 = 1) -> c.lui -> OK!

lui_uart:
    c.mv x13,x15
    c.jal uart_write_loop

####################
####################
## c.sw  instruction test, x0 - x15
## c.lw  instruction test, x8 - x15
## c.swsp  instruction test, x0 - x15
## c.lwsp  instruction test, x0 - x15
    c.li x1,1
    c.li x2,1
    c.li x3,1
    c.li x4,1
    c.li x5,1
```

```
575     c.li x6,1
576     c.li x7,1
577     c.li x8,1
578     c.li x9,1
579     c.li x10,1
580     c.li x11,1
581     c.li x12,1
582     c.li x13,1
583     c.li x14,1
584
585     c.lui x15,0x00001 # forma o endereço da ram, 0x1001_0000
586     c.addi x15,1
587     c.slli x15,16
588
589     c.sw x8,0(x15)
590     c.sw x9,4(x15)
591     c.sw x10,8(x15)
592     c.sw x11,12(x15)
593     c.sw x12,16(x15)
594     c.sw x13,20(x15)
595     c.sw x14,24(x15)
596     c.lw x8,0(x15)
597     c.lw x9,4(x15)
598     c.lw x10,8(x15)
599     c.lw x11,12(x15)
600     c.lw x12,16(x15)
601     c.lw x13,20(x15)
602     c.lw x14,24(x15)
603
604     c.add x9,x8
605     c.add x10,x9
606     c.add x11,x10
607     c.add x12,x11
608     c.add x13,x12
609     c.add x14,x13
610     c.addi x14,-6 # if (x14 = 1) -> c.sw , c.lw -> OK!
611
612 lw_sw_uart:
613     c.mv x2,x15
614     c.mv x13,x14
615     c.jal uart_write_loop
```

```
616
617     c.li x1,1
618     c.li x3,1
619     c.li x4,1
620     c.li x5,1
621     c.li x6,1
622     c.li x7,1
623     c.li x8,1
624     c.li x9,1
625     c.li x10,1
626     c.li x11,1
627     c.li x12,1
628     c.li x13,1
629     c.li x14,1
630     c.li x15,1
631
632     c.swsp x1,8(x2)
633     c.swsp x3,12(x2)
634     c.swsp x4,60(x2)
635     c.swsp x5,16(x2)
636     c.swsp x6,20(x2)
637     c.swsp x7,24(x2)
638     c.swsp x8,28(x2)
639     c.swsp x9,32(x2)
640     c.swsp x10,36(x2)
641     c.swsp x11,40(x2)
642     c.swsp x12,44(x2)
643     c.swsp x13,48(x2)
644     c.swsp x14,52(x2)
645     c.swsp x15,56(x2)
646     c.lwsp x1,8(x2)
647     c.lwsp x3,12(x2)
648     c.lwsp x4,60(x2)
649     c.lwsp x5,16(x2)
650     c.lwsp x6,20(x2)
651     c.lwsp x7,24(x2)
652     c.lwsp x8,28(x2)
653     c.lwsp x9,32(x2)
654     c.lwsp x10,36(x2)
655     c.lwsp x11,40(x2)
656     c.lwsp x12,44(x2)
```

```
657     c.lwsp x13,48(x2)
658     c.lwsp x14,52(x2)
659     c.lwsp x15,56(x2)
660 
661     c.add x3,x1
662     c.add x4,x3
663     c.add x5,x4
664     c.add x6,x5
665     c.add x7,x6
666     c.add x8,x7
667     c.add x9,x8
668     c.add x10,x9
669     c.add x11,x10
670     c.add x12,x11
671     c.add x13,x12
672     c.add x14,x13
673     c.add x15,x14
674     c.addi x15,-13 # if (x15 = 1) -> c.lwsp , c.swsp -> OK!
675 
676 lwsp_swsp_uart:
677     c.mv x13,x15
678     c.jal uart_write_loop
679 
680 line_feed_uart:
681     c.li x13,10
682     c.jal uart_send_linefeed
683 
684     loop_a:
685     c.nop
686     c.j loop_a
687 
688 ################################
689 #### Leitura de UART_CTRL, registrador de controle da UART
690     uart_read_ctrl:
691     c.lui x15,UART_HI
692     c.lw x14,0(x15)
693     c.jr x1
694 
695 ################################
696 #### Leitura de UART_RxT, dados recebidos pela UART
697     uart_read_loop:
```

```
    c.lui x15,UART_HI
loop1_b:
    c.lw x14,0(x15)
    c.andi x14,2 # bit 1 indica que está pronta para leitura, quando = '1'
    c.addi x14,-2
    c.bnez x14,loop1_b

    c.lw x13,UART_RxT_LO(x15) # x13 = dado lido
    c.jr x1

################################
#### Leitura de UART_RxT, dados recebidos pela UART
    uart_write_loop:
    c.lui x15,UART_HI
loop2_b:
    c.lw x14,0(x15)
    c.andi x14,8 # bit 1 indica que está pronta para enviar, quando = '1'
    c.addi x14,-8
    c.bnez x14,loop2_b

    c.addi x13,30
    c.addi x13,18 # ascii table offset
    c.sw x13,UART_RxT_LO(x15) # x13 = dado escrito
    c.jr x1

    uart_send_linefeed:
    c.lui x15,UART_HI
loop3_b:
    c.lw x14,0(x15)
    c.andi x14,8 # bit 1 indica que está pronta para enviar, quando = '1'
    c.addi x14,-8
    c.bnez x14,loop3_b

    c.sw x13,UART_RxT_LO(x15) # x13 = dado escrito
    c.jr x1
```

APÊNDICE B – DADOS DE SÍNTESE E IMPLEMENTAÇÃO

B.1 – NÚCLEO MULTICÍCLICO

```
DETAILED REPORTS: PLACE AND ROUTE REPORT

Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                   696 out of  18,224    3%
    Number used as Flip Flops:                 696
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                        961 out of   9,112   10%
    Number used as logic:                      960 out of   9,112   10%
      Number using O6 output only:             940
      Number using O5 output only:               0
      Number using O5 and O6:                   20
      Number used as ROM:                        0
    Number used as Memory:                       0 out of   2,176    0%
    Number used exclusively as route-thrus:      1
      Number with same-slice register load:      1
      Number with same-slice carry load:         0
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                   429 out of   2,278   18%
  Number of MUXCYs used:                        32 out of   4,556    1%
  Number of LUT Flip Flop pairs used:        1,435
    Number with an unused Flip Flop:           740 out of   1,435   51%
    Number with an unused LUT:                 474 out of   1,435   33%
    Number of fully used LUT-FF pairs:         221 out of   1,435   15%
    Number of slice register sites lost
      to control set restrictions:               0 out of  18,224    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
```

```
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                        69 out of     232   29%
    Number of LOCed IOBs:                        1 out of      69    1%

Specific Feature Utilization:
  Number of RAMB16BWERs:                        0 out of      32    0%
  Number of RAMB8BWERs:                         0 out of      64    0%
  Number of BUFIO2/BUFIO2_2CLKs:                0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:            0 out of      32    0%
  Number of BUFG/BUFGMUXs:                      1 out of      16    6%
    Number used as BUFGs:                       1
    Number used as BUFGMUX:                     0
  Number of DCM/DCM_CLKGENs:                    0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:                  0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:        0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:                  0 out of     248    0%
  Number of BSCANs:                             0 out of       4    0%
  Number of BUFHs:                              0 out of     128    0%
  Number of BUFPLLs:                            0 out of       8    0%
  Number of BUFPLL_MCBs:                        0 out of       4    0%
  Number of DSP48A1s:                           0 out of      32    0%
  Number of ICAPs:                              0 out of       1    0%
  Number of MCBs:                               0 out of       2    0%
  Number of PCILOGICSEs:                        0 out of       2    0%
  Number of PLL_ADVs:                           0 out of       2    0%
  Number of PMVs:                               0 out of       1    0%
  Number of STARTUPs:                           0 out of       1    0%
  Number of SUSPEND_SYNCs:                      0 out of       1    0%

---------------------------------------------------------------------------
  DETAILED REPORTS: Post-PAR Static Timing Report

  Timing summary:
---------------
```

```
Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 166532 paths, 0 nets, and 6954 connections

Design statistics:
   Minimum period:   9.829ns{1}   (Maximum frequency: 101.740MHz)
   Minimum input required time before clock:   7.970ns
   Minimum output required time after clock:   9.919ns
```

B.2 – NÚCLEO MULTICÍCLICO COM SISTEMA COMPLETO

```
DETAILED REPORTS: PLACE AND ROUTE REPORT

Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                   842 out of  18,224    4%
    Number used as Flip Flops:                 834
    Number used as Latches:                      8
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                      1,223 out of   9,112   13%
    Number used as logic:                    1,220 out of   9,112   13%
      Number using O6 output only:           1,074
      Number using O5 output only:              17
      Number using O5 and O6:                  129
      Number used as ROM:                        0
    Number used as Memory:                       1 out of   2,176    1%
      Number used as Dual Port RAM:              0
      Number used as Single Port RAM:            0
      Number used as Shift Register:             1
        Number using O6 output only:             1
        Number using O5 output only:             0
        Number using O5 and O6:                  0
    Number used exclusively as route-thrus:      2
      Number with same-slice register load:      1
```

```
      Number with same-slice carry load:         1
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                   518 out of   2,278   22%
  Number of MUXCYs used:                       160 out of   4,556    3%
  Number of LUT Flip Flop pairs used:        1,696
    Number with an unused Flip Flop:           872 out of   1,696   51%
    Number with an unused LUT:                 473 out of   1,696   27%
    Number of fully used LUT-FF pairs:         351 out of   1,696   20%
    Number of slice register sites lost
      to control set restrictions:               0 out of  18,224    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                         4 out of     232    1%
    Number of LOCed IOBs:                        4 out of       4  100%

Specific Feature Utilization:
  Number of RAMB16BWERs:                        16 out of      32   50%
  Number of RAMB8BWERs:                          0 out of      64    0%
  Number of BUFIO2/BUFIO2_2CLKs:                 0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:             0 out of      32    0%
  Number of BUFG/BUFGMUXs:                       1 out of      16    6%
    Number used as BUFGs:                        1
    Number used as BUFGMUX:                      0
  Number of DCM/DCM_CLKGENs:                     0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:                   0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:         0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:                   0 out of     248    0%
  Number of BSCANs:                              0 out of       4    0%
  Number of BUFHs:                               0 out of     128    0%
  Number of BUFPLLs:                             0 out of       8    0%
  Number of BUFPLL_MCBs:                         0 out of       4    0%
```

```
  Number of DSP48A1s:                          0 out of      32    0%
  Number of ICAPs:                             0 out of       1    0%
  Number of MCBs:                              0 out of       2    0%
  Number of PCILOGICSEs:                       0 out of       2    0%
  Number of PLL_ADVs:                          0 out of       2    0%
  Number of PMVs:                              0 out of       1    0%
  Number of STARTUPs:                          0 out of       1    0%
  Number of SUSPEND_SYNCs:                     0 out of       1    0%

---------------------------------------------------------------------
  DETAILED REPORTS: Post-PAR Static Timing Report

Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 571130 paths, 0 nets, and 8518 connections

Design statistics:
   Minimum period:   9.600ns{1}   (Maximum frequency: 104.167MHz)
   Minimum input required time before clock:   2.046ns
   Minimum output required time after clock:   8.919ns
```

B.3 – NÚCLEO PIPELINE

```
DETAILED REPORTS: PLACE AND ROUTE REPORT

Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                   975 out of  18,224    5%
    Number used as Flip Flops:                 975
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                      1,660 out of   9,112   18%
```

```
    Number used as logic:                   1,657 out of   9,112   18%
      Number using O6 output only:          1,490
      Number using O5 output only:             56
      Number using O5 and O6:                 111
      Number used as ROM:                       0
    Number used as Memory:                      0 out of   2,176    0%
    Number used exclusively as route-thrus:     3
      Number with same-slice register load:     1
      Number with same-slice carry load:        2
      Number with other load:                   0

Slice Logic Distribution:
  Number of occupied Slices:                  605 out of   2,278   26%
  Number of MUXCYs used:                      200 out of   4,556    4%
  Number of LUT Flip Flop pairs used:       2,102
    Number with an unused Flip Flop:        1,156 out of   2,102   54%
    Number with an unused LUT:                442 out of   2,102   21%
    Number of fully used LUT-FF pairs:        504 out of   2,102   23%
    Number of slice register sites lost
      to control set restrictions:              0 out of  18,224    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                       74 out of     232   31%
    Number of LOCed IOBs:                       2 out of      74    2%

Specific Feature Utilization:
  Number of RAMB16BWERs:                        2 out of      32    6%
  Number of RAMB8BWERs:                         0 out of      64    0%
  Number of BUFIO2/BUFIO2_2CLKs:                0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:            0 out of      32    0%
  Number of BUFG/BUFGMUXs:                      1 out of      16    6%
    Number used as BUFGs:                       1
    Number used as BUFGMUX:                     0
```

```
  Number of DCM/DCM_CLKGENs:                        0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:                      0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:            0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:                      0 out of     248    0%
  Number of BSCANs:                                 0 out of       4    0%
  Number of BUFHs:                                  0 out of     128    0%
  Number of BUFPLLs:                                0 out of       8    0%
  Number of BUFPLL_MCBs:                            0 out of       4    0%
  Number of DSP48A1s:                               0 out of      32    0%
  Number of ICAPs:                                  0 out of       1    0%
  Number of MCBs:                                   0 out of       2    0%
  Number of PCILOGICSEs:                            0 out of       2    0%
  Number of PLL_ADVs:                               0 out of       2    0%
  Number of PMVs:                                   0 out of       1    0%
  Number of STARTUPs:                               0 out of       1    0%
  Number of SUSPEND_SYNCs:                          0 out of       1    0%

--------------------------------------------------------------------------
DETAILED REPORTS: Post-PAR Static Timing Report

Timing summary:
---------------

Timing errors: 13  Score: 2540  (Setup/Max: 2540, Hold: 0)

Constraints cover 711077 paths, 0 nets, and 10618 connections

Design statistics:
   Minimum period:   9.963ns{1}   (Maximum frequency: 100.371MHz)
   Minimum input required time before clock:   9.390ns
   Minimum output required time after clock:  10.534ns
```

B.4 – NÚCLEO PIPELINE COM SISTEMA COMPLETO

```
DETAILED REPORTS: PLACE AND ROUTE REPORT

Slice Logic Utilization:
```

Number of Slice Registers:	1,137	out of	18,224	6%
Number used as Flip Flops:	1,129			
Number used as Latches:	8			
Number used as Latch-thrus:	0			
Number used as AND/OR logics:	0			
Number of Slice LUTs:	2,137	out of	9,112	23%
Number used as logic:	2,133	out of	9,112	23%
Number using O6 output only:	1,750			
Number using O5 output only:	73			
Number using O5 and O6:	310			
Number used as ROM:	0			
Number used as Memory:	1	out of	2,176	1%
Number used as Dual Port RAM:	0			
Number used as Single Port RAM:	0			
Number used as Shift Register:	1			
Number using O6 output only:	1			
Number using O5 output only:	0			
Number using O5 and O6:	0			
Number used exclusively as route-thrus:	3			
Number with same-slice register load:	0			
Number with same-slice carry load:	3			
Number with other load:	0			

Slice Logic Distribution:

Number of occupied Slices:	782	out of	2,278	34%
Number of MUXCYs used:	332	out of	4,556	7%
Number of LUT Flip Flop pairs used:	2,518			
Number with an unused Flip Flop:	1,485	out of	2,518	58%
Number with an unused LUT:	381	out of	2,518	15%
Number of fully used LUT-FF pairs:	652	out of	2,518	25%
Number of slice register sites lost to control set restrictions:	0	out of	18,224	0%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element.
The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.

```
IO Utilization:
  Number of bonded IOBs:                          5 out of     232    2%
    Number of LOCed IOBs:                         5 out of       5  100%

Specific Feature Utilization:
  Number of RAMB16BWERs:                         18 out of      32   56%
  Number of RAMB8BWERs:                           0 out of      64    0%
  Number of BUFIO2/BUFIO2_2CLKs:                  0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:              0 out of      32    0%
  Number of BUFG/BUFGMUXs:                        1 out of      16    6%
    Number used as BUFGs:                         1
    Number used as BUFGMUX:                       0
  Number of DCM/DCM_CLKGENs:                      0 out of       4    0%
  Number of ILOGIC2/ISERDES2s:                    0 out of     248    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:          0 out of     248    0%
  Number of OLOGIC2/OSERDES2s:                    0 out of     248    0%
  Number of BSCANs:                               0 out of       4    0%
  Number of BUFHs:                                0 out of     128    0%
  Number of BUFPLLs:                              0 out of       8    0%
  Number of BUFPLL_MCBs:                          0 out of       4    0%
  Number of DSP48A1s:                             0 out of      32    0%
  Number of ICAPs:                                0 out of       1    0%
  Number of MCBs:                                 0 out of       2    0%
  Number of PCILOGICSEs:                          0 out of       2    0%
  Number of PLL_ADVs:                             0 out of       2    0%
  Number of PMVs:                                 0 out of       1    0%
  Number of STARTUPs:                             0 out of       1    0%
  Number of SUSPEND_SYNCs:                        0 out of       1    0%

---------------------------------------------------------------------------
  DETAILED REPORTS: Post-PAR Static Timing Report

Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 2172693 paths, 0 nets, and 13251 connections
```

```
Design statistics:
   Minimum period:   9.991ns{1}   (Maximum frequency: 100.090MHz)
   Minimum input required time before clock:   2.016ns
   Minimum output required time after clock:   7.706ns
```

APÊNDICE C – DESCRIÇÃO VHDL - MEMÓRIA

O componente descrito em VHDL no arquivo RVC_dec.vhd está presente apenas na implementação do núcleo com pipeline. A RAM possuí duas descrições, uma com habilitação de escrita por byte utilizada na descrição do núcleo com pipeline, e uma sem habilitação de escrita por byte, na implementação do núcleo multicíclico. O arquivo de topo da memória também está separado por núcleo, devido a diferenças.

C.1 – DEC.VHD - MULTICÍCLICO

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  -- @file DEC.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2017
8  -- @version 1
9  -- Decodificador que serve de interface entre o núcleo e a Memória,
10 -- que se divide em uma ROM e uma RAM. Ambas descritas como memórias
11 -- de duas portas simétricas de 16 bits, compartilhando o mesmo sinal
12 -- de clock, e no caso da RAM, de habilitação de escrita também. Como
13 -- compartilham o mesmo barramento de dados, não podem escrever
14 -- simultaneamente no barramento, sendo controlada a lógica pelos
15 -- sinais cs_ROM e cs_RAM, após a decodificação dos 16 bits superiores
16 -- do endereço de entrada. Possuí um sinal que indica quando um dado
17 -- da memória qu esta sendo acessado não está pronto, wait_mem, que
18 -- tem um ciclo de laténcia, conforme o funcionamento de ambas as
19 -- memórias. Caso o sinal de entrada Valid_addr indique que o endereço
20 -- não épara ser utilizado, nenhum dos sinais cs_ROM ou cs_RAM são
21 -- habilitados, independente dos valores de addr_MEM[31:16]. De
22 -- acordo com o mapeamento de dispositivos IO diretamente na memória,
23 -- esse componente também decodifica o acesso a dispositivos IO,
24 -- enviando o sinal de habilitação csIO para a saída.
25 entity DEC is
26   generic(n : integer := 32); -- tamanho dos dados
27     port (rst : in std_logic; -- sinal de reset do ssitema
28         clk : in std_logic; -- sinal de clock do sistema
```

```
29        we_l : in std_logic; -- habilitação de escrita
30        Valid_addr : in std_logic; -- indica quando o endereço no barramento é
              valido
31        csIO : out std_logic; -- habilitação da sessão IO
32        wait_mem : out std_logic; -- sinal wait, indicando que o dado da memória
              não esta disponível ainda
33        addr_MEM : in std_logic_vector(n-1 downto 0); -- endereço de acesso
34        data_MEM : inout std_logic_vector(n-1 downto 0));-- dado de 4 bytes de
              entrada ou saída da memória, buffer tri state
35  end entity DEC;
36
37  architecture behavior of DEC is
38
39  ----------------------------------------
40  ---- BLOCK RAM(s)
41  ------------------------
42  -------- ROM
43    constant n_dataAB_ROM : integer := 16;
44    constant n_addrAB_ROM : integer := 13;
45    constant MSB_ROM : std_logic_vector(15 downto 0) := x"0000"; -- constante para
          verificar o MSByte do endereço, utilizado para decodificar o uso da ROM
46
47      signal cs_ROM : std_logic; -- chip select da ROM, ativo se a condição de
          MSB_ROM for verdadeira
48    signal addra_ROM : std_logic_vector(n_addrAB_ROM-1 downto 0);
49      signal addrb_ROM : std_logic_vector(n_addrAB_ROM-1 downto 0);
50      signal douta_ROM : std_logic_vector(n_dataAB_ROM-1 downto 0);
51      signal doutb_ROM : std_logic_vector(n_dataAB_ROM-1 downto 0);
52    signal data_ROM : std_logic_vector((n_dataAB_ROM*2)-1 downto 0); -- dado de sa
          ída da ROM
53
54      COMPONENT ROM
55          generic(n_dataAB : integer;
56                  n_addrAB : integer);
57          PORT(clkab : in std_logic;
58              addra : in std_logic_vector(n_addrAB-1 downto 0);
59              douta : out std_logic_vector(n_dataAB-1 downto 0);
60              addrb : in std_logic_vector(n_addrAB-1 downto 0);
61              doutb : out std_logic_vector(n_dataAB-1 downto 0));
62      END COMPONENT ROM;
63
```

```
64 ------------------------
65 -------- RAM
66    constant n_dataAB_RAM : integer := 16;
67    constant n_addrAB_RAM : integer := 13;
68    constant MSB_RAM  : std_logic_vector(15 downto 0) := x"1001"; -- constante
          para verificar o MSByte do endereço, utilizado para decodificar o uso da
          RAM
69
70    signal cs_RAM : std_logic; -- chip select da RAM, ativo se a condição de
          MSB_RAM for verdadeira
71  signal oe_ram : std_logic; -- sinal de habilitação de saída da RAM
72  signal addra_RAM: std_logic_vector(n_addrAB_RAM-1 downto 0);
73  signal addrb_RAM: std_logic_vector(n_addrAB_RAM-1 downto 0);
74  signal dina_RAM : std_logic_vector(n_dataAB_RAM-1 downto 0);
75    signal dinb_RAM : std_logic_vector(n_dataAB_RAM-1 downto 0);
76    signal douta_RAM: std_logic_vector(n_dataAB_RAM-1 downto 0);
77    signal doutb_RAM: std_logic_vector(n_dataAB_RAM-1 downto 0);
78    signal data_RAM : std_logic_vector((n_dataAB_RAM*2)-1 downto 0); -- dado
          lido da RAM
79    signal weab : std_logic_vector(0 downto 0); -- habilitação de escrita da
          porta A da RAM
80
81  COMPONENT RAM
82    generic(n_dataAB : integer;
83         n_addrAB : integer);
84    PORT(clkab  : in std_logic;
85      weab : in std_logic_vector(0 downto 0);
86      addra  : in std_logic_vector(n_addrAB-1 downto 0);
87      dina : in std_logic_vector(n_dataAB-1 downto 0);
88      douta  : out std_logic_vector(n_dataAB-1 downto 0);
89      addrb  : in std_logic_vector(n_addrAB-1 downto 0);
90      dinb : in std_logic_vector(n_dataAB-1 downto 0);
91      doutb  : out std_logic_vector(n_dataAB-1 downto 0));
92    END COMPONENT RAM;
93 ----------------------------------------
94
95    constant MSB_IO : std_logic_vector(15 downto 0) := x"FFFF"; -- constante
          para verificar o MSByte do endereço, utilizado para decodificar o uso de
          IOs
96
```

```
97     signal count : std_logic; -- conta o número de ciclos em que wait = '1' (no
           caso, 1)
98
99  begin
100
101 -- habilitação de escrita da RAM
102     weab <= "1" when (we_l = '0' AND cs_RAM = '0') else "0";
103
104 -- habilitação de saída da RAM
105   oe_ram <= '1' when (cs_RAM = '0' AND we_l = '1') else '0';
106
107 -- sinais de chip select, ativos em '0'
108     csIO <= '0' when (addr_MEM(31 downto 16) = MSB_IO AND rst = '0') else
109                 '1';
110
111   cs_RAM <= '0' when (addr_MEM(31 downto 16) = MSB_RAM AND Valid_addr = '1' AND
          rst = '0') else
112                 '1';
113
114   cs_ROM <= '0' when (addr_MEM(31 downto 16) = MSB_ROM AND Valid_addr = '1' AND
          rst = '0') else
115                 '1';
116
117 -- Dado de saída da ROM, e endereços de entrada
118   addra_ROM <= addr_MEM(n_addrAB_ROM-1+1 downto 1);
119   addrb_ROM <= addr_MEM(n_addrAB_ROM-1+1 downto 1) + 1; -- +2 (offset de 1 bit)
120
121   data_ROM <= doutb_ROM&douta_ROM;
122
123 -- Dado de saída e entrada da RAM, e endereços de entrada
124   dina_RAM <= data_MEM(n_dataAB_RAM-1 downto 0);
125   dinb_RAM <= data_MEM(31 downto n_dataAB_RAM);
126
127   addra_RAM <= addra_ROM;
128   addrb_RAM <= addrb_ROM;
129
130   data_RAM <= doutb_RAM&douta_RAM;
131
132 -- Entrada\Saída de dados
133     data_MEM(n-1 downto 0) <= data_ROM when (cs_ROM = '0') else
134                               (others => 'Z');
```

```
135
136   data_MEM(n-1 downto 0) <= data_RAM when (cs_RAM = '0' AND we_l = '1') else
137                             (others => 'Z');
138
139 --------------------------------
140 -- lógica do sinal wait_mem, duração de 1 ciclo
141     wait_mem <= '1' when ((oe_ram = '1' OR cs_ROM = '0') AND count = '0') else
142                 '0';
143
144     process(clk , rst , oe_ram , cs_ROM)
145     variable pause : std_logic; -- evita que o count tenha efeito por um ciclo a
            mais
146     begin
147         if (rising_edge(clk)) then
148             if (rst = '1') then
149                 count <= '0';
150                 pause := '0';
151
152             elsif ((oe_ram = '1' OR cs_ROM = '0') AND pause = '0') then --
                    leitura
153                 count <= '1';
154                 pause := '1';
155
156             else
157                 count <= '0';
158                 pause := '0';
159
160             end if;
161         end if;
162     end process;
163
164 --------------------------------
165 -- port map
166   -- instanciação da ROM
167   ROM_i : ROM
168   GENERIC MAP (
169   n_addrAB => n_addrAB_ROM ,
170   n_dataAB => n_dataAB_ROM)
171   PORT MAP (
172   clkab => clk ,
173   addra => addra_ROM ,
```

```
174   douta => douta_ROM ,
175   addrb => addrb_ROM ,
176   doutb => doutb_ROM
177   );
178
179   -- instanciação da RAM
180     RAM_i : RAM
181     GENERIC MAP (
182     n_dataAB => n_dataAB_RAM ,
183     n_addrAB => n_addrAB_RAM
184     )
185     PORT MAP (
186     clkab  => clk ,
187     weab  => weab ,
188   addra  => addra_RAM ,
189   douta  => douta_RAM ,
190   dina => dina_RAM ,
191   addrb  => addrb_RAM ,
192   doutb  => doutb_RAM ,
193   dinb => dinb_RAM
194   );
195
196 end architecture behavior;
```

C.2 – DEC.VHD - PIPELINE

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  -- @file DEC.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2018
8  -- @version 1
9  -- Decodificador que serve de interface entre o núcleo e a Memória,
10 -- que se divide em uma ROM e uma RAM. Ambas descritas como memórias
11 -- de duas portas simétricas de 16 bits, compartilhando o mesmo sinal
12 -- de clock. Na RAM se tem ainda o sinal de escrita, que na porta A
13 -- éseparado por byte, e na porta B éunificado para todo o dado
14 -- de entrada, sendo que quando se escreve na porta B, a porta A
```

```
15 -- também éescrita, pois a porta A éa única utilizada para
16 -- escritas de byte ou half-word. Como as memórias compartilham o
17 -- mesmo barramento de dados, não podem escrever simultaneamente no
18 -- barramento, sendo controlada a lógica pelos sinais cs_ROM e cs_RAM,
19 -- após a decodificação dos 16 bits superiores do endereço de entrada.
20 -- Possuí um sinal que indica quando um dado de instrução não está
21 -- pronto, wait_instr, e um para dados, wait_data, ambos com uma
22 -- laténcia de um ciclo de acesso, conforme o acesso síncrono as
23 -- memórias para leitura. Caso o sinal de entrada Valid_addr indique
24 -- que o endereço não épara ser utilizado, nenhum dos sinais cs_ROM ou
25 -- cs_RAM são habilitados, independente dos valores de addr_MEM[31:16].
26 -- De acordo com o mapeamento de dispositivos IO diretamente na memória,
27 -- esse componente também decodifica o acesso a dispositivos IO,
28 -- enviando o sinal de habilitação csIO para a saída. O bloco RVC_dec
29 -- decodifica/expande instruções RVC de 16 bits, para as correspondentes
30 -- de 32 bits da base E, caso necessário pela verificação dos 2LSB do
31 -- dado de saída da ROM. Outro caso em que a unidade RVC_dec ignora a
32 -- decodificação, além do caso de a instrução não ser compressa, é
33 -- na leitura de dados constantes de 32 bits (seção .rodata) da ROM.
34 entity DEC is
35   generic(n : integer := 32); -- tamanho dos dados
36     port (rst : in std_logic; -- sinal de reset do ssitema
37         clk : in std_logic; -- sinal de clock do sistema
38     we : in std_logic; -- habilitação de escrita
39     re   : in std_logic;       -- habilitação de leitura
40         sb_en : in std_logic; -- sinal store byte enable
41         sh_en : in std_logic; -- sinal store half enable
42         Valid_addr : in std_logic; -- sinal de validade do endereço do barramento
               , quando '1' = invalido
43         csIO : out std_logic; -- habilitação da sessão IO
44         wait_data : out std_logic; -- sinal wait, indicando que o dado da memória
                não esta disponível ainda
45     wait_instr : out std_logic; -- sinal wait, indicando que a instr. da memória
            não esta disponível
46         rvc : out std_logic; -- sinal que indica que a instr. éda extensão rvc
47         addr_MEM : in std_logic_vector(n-1 downto 0); -- endereço de acesso
48         data_MEM : inout std_logic_vector(n-1 downto 0));-- dado de 4 bytes de
              entrada ou saída da memória, buffer tri state
49 end entity DEC;
50
51 architecture behavior of DEC is
```

```
52
53   COMPONENT RVC_dec
54     generic(n  : integer);
55     port(data_in : in std_logic_vector(n-1 downto 0);
56       re_rodata: in std_logic;
57            rvc : out std_logic;
58            data_out : out std_logic_vector(n-1 downto 0));
59     end COMPONENT RVC_dec;
60
61 ----------------------------------------
62 ---- BLOCK RAM(s)
63 --------------------------------
64 -------- ROM
65     constant n_dataAB_ROM : integer := 16;
66     constant n_addrAB_ROM : integer := 13;
67     constant MSB_ROM : std_logic_vector(15 downto 0) := x"0000"; -- constante
           para verificar o MSByte do endereço, utilizado para decodificar o uso da
           ROM
68
69   signal re_rodata : std_logic; -- indica quando o acesso a ROM épara leitura de
         dados e não instr, trocando o sinal wait_instr pelo wait_data
70     signal cs_ROM : std_logic; -- chip select da ROM, ativo se a condição de
           MSB_ROM for verdadeira
71     signal addra_ROM : std_logic_vector(n_addrAB_ROM-1 downto 0);
72     signal addrb_ROM : std_logic_vector(n_addrAB_ROM-1 downto 0);
73     signal douta_ROM : std_logic_vector(n_dataAB_ROM-1 downto 0);
74     signal doutb_ROM : std_logic_vector(n_dataAB_ROM-1 downto 0);
75     signal data_ROM : std_logic_vector(n-1 downto 0); -- dado de saída da ROM
76
77     COMPONENT ROM
78         generic(n_dataAB : integer;
79                 n_addrAB : integer);
80         PORT(clkab : in std_logic;
81             addra : in std_logic_vector(n_addrAB-1 downto 0);
82             douta : out std_logic_vector(n_dataAB-1 downto 0);
83             addrb : in std_logic_vector(n_addrAB-1 downto 0);
84             doutb : out std_logic_vector(n_dataAB-1 downto 0));
85     END COMPONENT ROM;
86
87 --------------------------------
88 -------- RAM
```

```
89    constant n_dataAB_RAM: integer := 16;
90    constant n_addrAB_RAM: integer := 13;
91   constant MSB_RAM : std_logic_vector(15 downto 0) := x"1001"; -- constante para
         verificar o MSByte do endereço, utilizado para decodificar o uso da RAM
92
93    signal cs_RAM : std_logic; -- chip select da RAM, ativo se a condição de
         MSB_RAM for verdadeira
94    signal oe_ram : std_logic;
95    signal wea : std_logic_vector(1 downto 0); -- byte enable da porta A da BRAM
96   signal web : std_logic_vector(0 downto 0); -- byte enable da porta B da BRAM
97   signal addra_RAM: std_logic_vector(n_addrAB_RAM-1 downto 0);
98    signal addrb_RAM: std_logic_vector(n_addrAB_RAM-1 downto 0);
99    signal dina_RAM : std_logic_vector(n_dataAB_RAM-1 downto 0);
100   signal dinb_RAM : std_logic_vector(n_dataAB_RAM-1 downto 0);
101   signal douta_RAM: std_logic_vector(n_dataAB_RAM-1 downto 0);
102   signal doutb_RAM: std_logic_vector(n_dataAB_RAM-1 downto 0);
103
104   COMPONENT RAM
105   generic(n_dataAB : integer;
106       n_addrAB : integer);
107   port(clkab : in std_logic;
108     addra : in std_logic_vector(n_addrAB-1 DOWNTO 0);
109     dina : in std_logic_vector(n_dataAB-1 DOWNTO 0);
110     douta : out std_logic_vector(n_dataAB-1 DOWNTO 0);
111     wea  : in std_logic_vector(1 downto 0);
112     addrb : in std_logic_vector(n_addrAB-1 DOWNTO 0);
113     dinb : in std_logic_vector(n_dataAB-1 DOWNTO 0);
114     doutb : out std_logic_vector(n_dataAB-1 DOWNTO 0);
115     web  : in std_logic_vector(0 DOWNTO 0)
116     );
117   END COMPONENT RAM;
118 ----------------------------------------
119
120   constant MSB_IO : std_logic_vector(15 downto 0) := x"FFFF"; -- constante
         para verificar o MSByte do endereço, utilizado para decodificar o uso de
         IOs
121
122   signal instr : std_logic_vector(n-1 downto 0); -- instrução de saída do dec
         para instr RVC
123
124  signal addra_mem : std_logic_vector(n_addrAB_RAM-1 downto 0);
```

```
125   signal addrb_mem : std_logic_vector(n_addrAB_RAM-1 downto 0);
126
127     signal count_wait_data : std_logic;
128     signal count_wait_instr : std_logic;
129
130 begin
131
132 -- sinais de chip select, ativos em '0'
133     csIO <= '1' when (rst = '1') else
134                 '0' when (addr_MEM(31 downto 16) = MSB_IO AND Valid_addr = '0')
                        else '1';
135
136   cs_RAM <= '1' when (rst = '1') else
137                 '0' when (addr_MEM(31 downto 16) = MSB_RAM AND Valid_addr = '0')
                        else '1';
138
139   cs_ROM <= '1' when (rst = '1') else
140                 '0' when (addr_MEM(31 downto 16) = MSB_ROM AND Valid_addr = '0')
                        else '1';
141
142   re_rodata <= '1' when (cs_ROM = '0' AND re = '1') else '0';
143
144 -- byte enable e output enable da RAM
145   wea <= "01" when (cs_RAM = '0' AND we = '1' AND sb_en = '1' AND sh_en = '0'
          AND addr_MEM(0) = '0') else --SB
146       "10" when (cs_RAM = '0' AND we = '1' AND sb_en = '1' AND sh_en = '0' AND
              addr_MEM(0) = '1') else --SB
147       "11" when (cs_RAM = '0' AND we = '1') else -- AND sb_en = '0' AND sh_en =
              '1') else --> SH e SW
148       "00";
149
150   web <=  "1" when (cs_RAM = '0' AND we = '1' AND sb_en = '0' AND sh_en = '0')
          else
151       "0";
152
153     oe_ram <= '1' when (cs_RAM = '0' AND we = '0') else '0';
154
155 -- Endereço de entrada da ROM e da RAM (são iguais), assim como n_addrAB_ROM =
      n_addrAB_RAM
156   addra_mem <= addr_MEM(n_addrAB_RAM-1+1 downto 1);
157   addrb_mem <= addr_MEM(n_addrAB_RAM-1+1 downto 1) + 1; -- +2 (offset de 1 bit)
```

```
158
159 -- Endereço de entrada da RAM
160   addra_RAM <= addra_mem;
161   addrb_RAM <= addrb_mem;
162
163 -- Dado de saída da ROM, e endereços de entrada
164     addra_ROM <= addra_mem;--addr_MEM(n_addrAB_ROM-1+1 downto 1);
165     addrb_ROM <= addrb_mem;--addr_MEM(n_addrAB_ROM-1+1 downto 1) + 1; -- +2 (
          offset de 1 bit)
166
167     data_ROM <= doutb_ROM&douta_ROM;
168
169 -- Entrada de dados da RAM
170   dina_RAM(15 downto 8)<= data_MEM(15 downto 8) when (addr_MEM(0) = '0') else
171               data_MEM(7 downto 0);
172
173   dina_RAM(7 downto 0) <= data_MEM(7 downto 0); -- não interessa para addr_MEM
        (0) = '1'
174
175   dinb_RAM <= data_MEM(31 downto 16);
176
177 -- Saída de dados
178   -- ROM
179   data_MEM(n-1 downto 8)  <= instr(n-1 downto 8) when (cs_ROM = '0') else
180               (others => 'Z');
181
182   data_MEM(7 downto 0) <= instr(7 downto 0) when (cs_ROM = '0' AND addr_MEM(0) =
        '0') else
183               instr(15 downto 8) when (cs_ROM = '0' AND addr_MEM(0) = '1') else
184               (others => 'Z');
185
186   -- RAM
187   data_MEM(n-1 downto 8) <= doutb_RAM&douta_RAM(n_dataAB_RAM-1 downto 8) when (
        oe_ram = '1') else
188                    (others => 'Z');
189
190   data_MEM(7 downto 0) <=  douta_RAM(7 downto 0)  when (oe_ram = '1' AND
        addr_MEM(0) = '0') else
191               douta_RAM(15 downto 8)  when (oe_ram = '1' AND addr_MEM(0) = '1')
                      else
192               (others => 'Z');
```

```
193
194 --------------------------------
195 -- lógica do sinal wait_data
196     wait_data <='1' when ((oe_ram = '1' OR re_rodata = '1') AND count_wait_data
        = '0') else
197             '0';
198
199     process(clk , rst , oe_ram , re_rodata)
200     variable pause : std_logic;
201     begin
202         if (rising_edge(clk)) then
203             if (rst = '1') then
204                 count_wait_data <= '0';
205                 pause := '0';
206
207             elsif ((oe_ram = '1' OR re_rodata = '1') AND pause = '0') then --
                    leitura
208                 count_wait_data <= '1';
209                 pause := '1'; -- laténcia de apenas 1 ciclo no acesso de leitura
                        da memória
210
211             else
212                 count_wait_data <= '0';
213                 pause := '0';
214
215             end if;
216         end if;
217     end process;
218
219 --------------------------------
220 -- lógica do sinal wait_instr
221     wait_instr <= '1' when (cs_ROM = '0' AND count_wait_instr = '0' AND
        re_rodata = '0') else
222                 '0';
223
224     process(clk , rst , cs_ROM , re_rodata)
225     variable pause : std_logic;
226     begin
227         if (rising_edge(clk)) then
228             if (rst = '1') then
229                 count_wait_instr <= '0';
```

```
230                 pause := '0';
231
232             elsif (cs_ROM = '0' AND pause = '0' AND re_rodata = '0') then --
                    leitura
233                 count_wait_instr <= '1';
234                 pause := '1'; -- laténcia de apenas 1 ciclo no acesso de leitura
                        da memória
235
236             else
237                 count_wait_instr <= '0';
238                 pause := '0';
239
240             end if;
241         end if;
242     end process;
243
244 --------------------------------
245 -- port map
246  RVC_dec_i : RVC_dec
247  GENERIC MAP (
248  n => n)
249  PORT MAP (
250  data_in => data_ROM ,
251  re_rodata=> re_rodata ,
252     rvc => rvc ,
253     data_out => instr);
254
255     ROM_i : ROM
256     GENERIC MAP (
257     n_addrAB => n_addrAB_ROM ,
258     n_dataAB => n_dataAB_ROM)
259     PORT MAP (
260     clkab => clk ,
261     addra => addra_ROM ,
262     douta => douta_ROM ,
263     addrb => addrb_ROM ,
264     doutb => doutb_ROM
265     );
266
267     RAM_i : RAM
268     GENERIC MAP (
```

```
269     n_addrAB => n_addrAB_RAM ,
270     n_dataAB => n_dataAB_RAM)
271     PORT MAP (
272     clkab => clk ,
273   addra => addra_RAM ,
274   dina => dina_RAM ,
275   douta => douta_RAM ,
276   wea  => wea ,
277   addrb => addrb_RAM ,
278   dinb => dinb_RAM ,
279   doutb => doutb_RAM ,
280   web  => web
281   );
282
283 end architecture behavior;
```

C.3 – ROM.VHD

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4 use ieee.std_logic_unsigned.all;
 5 use ieee.std_logic_textio.all;
 6 use std.textio.all;
 7
 8 entity ROM is
 9     generic(n_dataAB : integer;
10             n_addrAB : integer);
11     port(clkab : in std_logic;
12         addra : in std_logic_vector(n_addrAB-1 DOWNTO 0);
13         douta : out std_logic_vector(n_dataAB-1 DOWNTO 0);
14         addrb : in std_logic_vector(n_addrAB-1 DOWNTO 0);
15         doutb : out std_logic_vector(n_dataAB-1 DOWNTO 0));
16 end entity ROM;
17
18 architecture behavior of ROM is
19
20     constant DATA_WIDTH_AB : integer := n_dataAB;
21     constant ADDR_WIDTH_AB : integer := n_addrAB;
22
```

```
23      type mem is ARRAY (0 to (2**ADDR_WIDTH_AB)-1) of std_logic_vector(
            DATA_WIDTH_AB-1 downto 0);
24
25      -- path to .hex file
26      constant filename : string :="path_to_hexfile/text.hex";
27
28      --- file read
29      impure function InitRomFromFile (RomFileName : in string) return mem is
30          FILE romfile : text is in RomFileName;
31          variable RomFileLine : line;
32          variable rom : mem;
33      begin
34          for i in mem'range loop
35              readline(romfile, RomFileLine);
36              hread(RomFileLine, rom(i));
37          end loop;
38          return rom;
39      end function;
40
41      signal rom : mem := InitRomFromFile(filename);
42
43  begin
44
45      --port A read
46      process(clkab)
47      begin
48          if(rising_edge(clkab)) then
49
50              douta <= rom(to_integer(unsigned(addra)));
51
52          end if;
53      end process;
54
55      -- port B read
56      process(clkab)
57      begin
58          if(rising_edge(clkab)) then
59
60              doutb <= rom(to_integer(unsigned(addrb)));
61
62          end if;
```

```
63    end process;
64
65 end architecture;
```

C.4 – RAM.VHD - MULTICÍCLICO

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4 use ieee.std_logic_unsigned.all;
 5 use ieee.std_logic_textio.all;
 6 use std.textio.all;
 7
 8 entity RAM is
 9   generic(n_dataAB : integer;
10       n_addrAB : integer);
11   PORT(clkab  : in std_logic;
12     weab : in std_logic_vector(0 downto 0);
13     addra  : in std_logic_vector(n_addrAB-1 downto 0);
14     dina : in std_logic_vector(n_dataAB-1 downto 0);
15     douta  : out std_logic_vector(n_dataAB-1 downto 0);
16     addrb  : in std_logic_vector(n_addrAB-1 downto 0);
17     dinb : in std_logic_vector(n_dataAB-1 downto 0);
18     doutb  : out std_logic_vector(n_dataAB-1 downto 0));
19 end entity RAM;
20
21 architecture behavior of RAM is
22
23     constant DATA_WIDTH_AB : integer := n_dataAB;
24     constant ADDR_WIDTH_AB : integer := n_addrAB;
25
26     type mem is ARRAY (0 to (2**ADDR_WIDTH_AB)-1) of std_logic_vector(
           DATA_WIDTH_AB-1 downto 0);
27
28   -- path to .hex file
29   constant filename  : string :="path_to_hexfile/sdata.hex";
30
31   --- file read
32   impure function InitRamFromFile (RamFileName : in string) return mem is
33     FILE ramfile   : text is in RamFileName;
```

```
34    variable RamFileLine : line;
35    variable ram   : mem;
36  begin
37    for i in mem'range loop
38      readline(ramfile, RamFileLine);
39      hread(RamFileLine, ram(i));
40    end loop;
41    return ram;
42  end function;
43
44    signal ram : mem := InitRamFromFile(filename);
45
46 begin
47
48  process(clkab) -- port A and B, write first
49    begin
50        if(rising_edge(clkab)) then
51
52            if(weab = "1") then
53        ram(to_integer(unsigned(addra))) <= dina;
54        ram(to_integer(unsigned(addrb))) <= dinb;
55            end if;
56
57      douta <= ram(to_integer(unsigned(addra)));
58      doutb <= ram(to_integer(unsigned(addrb)));
59
60        end if;
61    end process;
62
63 end architecture;
```

C.5 – RAM.VHD - PIPELINE

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use ieee.std_logic_unsigned.all;
5 use ieee.std_logic_textio.all;
6 use std.textio.all;
7
```

```
 8 entity RAM is
 9     generic(n_dataAB : integer;
10             n_addrAB : integer);
11     port(clkab : in std_logic;
12     addra : in std_logic_vector(n_addrAB-1 DOWNTO 0);
13     dina : in std_logic_vector(n_dataAB-1 DOWNTO 0);
14     douta : out std_logic_vector(n_dataAB-1 DOWNTO 0);
15     wea  : in std_logic_vector(1 downto 0);
16     addrb : in std_logic_vector(n_addrAB-1 DOWNTO 0);
17     dinb : in std_logic_vector(n_dataAB-1 DOWNTO 0);
18     doutb : out std_logic_vector(n_dataAB-1 DOWNTO 0);
19     web  : in std_logic_vector(0 DOWNTO 0)
20     );
21 end entity RAM;
22
23 architecture behavior of RAM is
24
25     constant DATA_WIDTH_AB : integer := n_dataAB;
26     constant ADDR_WIDTH_AB : integer := n_addrAB;
27
28   constant DATA_half : integer := n_dataAB/2;
29
30     type mem is ARRAY (0 to (2**ADDR_WIDTH_AB)-1) of std_logic_vector(
          DATA_WIDTH_AB-1 downto 0);
31
32     -- path to .hex file
33     constant filename : string :="path_to_hexfile/sdata.hex";
34
35     --- file read
36     impure function InitRamFromFile (RamFileName : in string) return mem is
37         FILE ramfile : text is in RamFileName;
38         variable RamFileLine : line;
39         variable ram : mem;
40     begin
41         for i in mem'range loop
42             readline(ramfile, RamFileLine);
43             hread(RamFileLine, ram(i));
44         end loop;
45         return ram;
46     end function;
47
```

```
48      shared variable ram : mem := InitRamFromFile(filename);
49
50  begin
51      --port A read
52      process(clkab)
53      begin
54          if(rising_edge(clkab)) then
55
56        if(wea(1) = '1') then
57          ram(to_integer(unsigned(addra)))(DATA_WIDTH_AB-1 downto DATA_half) :=
                dina(DATA_WIDTH_AB-1 downto DATA_half);
58        end if;
59        if(wea(0) = '1') then
60          ram(to_integer(unsigned(addra)))(DATA_half-1 downto 0) := dina(DATA_half
                -1 downto 0);
61        end if;
62
63              douta <= ram(to_integer(unsigned(addra)));
64
65          end if;
66      end process;
67
68      -- port B read
69      process(clkab)
70      begin
71          if(rising_edge(clkab)) then
72
73        if(web = "1") then
74          ram(to_integer(unsigned(addrb))) := dinb;
75        end if;
76
77        doutb <= ram(to_integer(unsigned(addrb)));
78
79          end if;
80      end process;
81
82  end architecture;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity RVC_dec is
5    generic(n  : integer);
6    port(data_in : in std_logic_vector(n-1 downto 0);
7      re_rodata : in std_logic;
8          rvc : out std_logic;
9          data_out : out std_logic_vector(n-1 downto 0));
10 end entity RVC_dec;
11
12 architecture behavior of RVC_dec is
13
14     -- sinal que indica que a instr. édo tipo RVC
15     signal rvc_t : std_logic;
16
17     -- campos rs2, rs1 e rd da instr. estendida
18     signal rs2 : std_logic_vector(4 downto 0);
19     signal rs1 : std_logic_vector(4 downto 0);
20     signal rd : std_logic_vector(4 downto 0);
21
22     -- rvc_fctop = funct3 & opcode, da instr RVC (16bits), campos [15:13]&[1:0]
23     signal rvc_fctop : std_logic_vector(4 downto 0);
24     signal fct3 : std_logic_vector(2 downto 0);
25     signal fct2 : std_logic_vector(1 downto 0);
26     signal fct1 : std_logic_vector(1 downto 0);
27     signal opc : std_logic_vector(1 downto 0);
28
29   -- c.ADD, c.MV, c.JALR e c.JR possuem o mesmo rvc_fctop, logo se utiliza
30   -- sinais extra para decodificação, o mesmo se aplica para c.LUI com c.
       ADDI16SP
31     signal c_jr : std_logic;
32     signal c_jalr : std_logic;
33     signal lui_en : std_logic;
34
35     -- imediato da instr RVC
36     signal shamt : std_logic_vector(4 downto 0);
37     signal sig_or_zero : std_logic;
38     signal mux_imm_zero : std_logic_vector(1 downto 0);
39     signal mux_imm_sig : std_logic_vector(1 downto 0);
40     signal imm_rvc : std_logic_vector(11 downto 0);
```

```
41    signal imm_field : std_logic_vector(12 downto 2);
42    signal imm_zeroext : std_logic_vector(11 downto 0);
43    signal imm_sigext : std_logic_vector(11 downto 0);
44
45 begin
46   -- ignora a verificação do tipo de instr., caso seja leitura de dados e não
         instr.
47    rvc_t <= '1' when (data_in(1 downto 0) /= "11" AND re_rodata = '0') else
48               '0';
49   rvc <= rvc_t;
50
51    rvc_fctop <= data_in(15 downto 13)&data_in(1 downto 0);
52    fct3 <= rvc_fctop(4 downto 2);
53    fct2 <= data_in(11 downto 10);
54    fct1 <= data_in(6 downto 5);
55    opc <= rvc_fctop(1 downto 0);
56
57    c_jalr <= '1' when (data_in(6 downto 2) = "00000" AND data_in(12) = '1')
          else
58               '0';
59
60    c_jr <= '1' when (data_in(6 downto 2) = "00000") else
61               '0';
62
63    lui_en <= '0' when (data_in(10 downto 7) = "0010") else
64               '1';
65
66    ----------------------------
67    -- instr. de saída
68
69    data_out(31) <= data_in(31) when (rvc_t = '0') else -- RVE Instr
70                    '0' when (fct3 = "100" OR rvc_fctop = "00010") else
71                    imm_rvc(11);
72
73    data_out(30) <= data_in(30) when (rvc_t = '0') else -- RVE Instr
74                    '1' when (rvc_fctop = "10001" AND (fct2 = "01" OR (fct2 = "11"
                          AND fct1 = "00"))) else -- SRAI, SUB
75                    '0' when (rvc_fctop = "10001" OR rvc_fctop = "00010" OR (
                             rvc_fctop = "10010" AND c_jr = '0')) else -- ADD, MV
76                    imm_rvc(11) when (rvc_fctop = "01101" AND lui_en = '1') else
                             -- LUI
```

```
77                  imm_rvc(10);
78
79      data_out(29 downto 25) <= data_in(29 downto 25) when (rvc_t = '0') else --
            RVE Instr
80                              (others => imm_rvc(11)) when (rvc_fctop = "01101"
                                    AND lui_en = '1') else -- LUI
81                              (others => '0') when ((rvc_fctop = "10001" AND fct2
                                     /= "10") OR rvc_fctop = "00010" OR (rvc_fctop =
                                     "10010" AND c_jr = '0')) else -- ADD, MV
82                              imm_rvc(9 downto 5);
83
84      data_out(24 downto 20) <= data_in(24 downto 20) when (rvc_t = '0') else --
            RVE Instr
85                              imm_rvc(4 downto 1)&imm_rvc(11) when (rvc_fctop = "
                                    00101" OR rvc_fctop = "10101") else -- JAL
86                              imm_rvc(11)&imm_rvc(11 downto 8) when (rvc_fctop =
                                    "01101" AND lui_en = '1') else -- LUI
87                              shamt when (rvc_fctop = "00010") else -- SLLI
88                              rs2 when (fct3(2 downto 1) = "11" OR rvc_fctop = "
                                    10010" OR (rvc_fctop = "10001" AND fct2 = "11"))
                                     else -- sub, add, mv, and, or, xor ,sw , beq/
                                    bne
89                              imm_rvc(4 downto 0);
90
91      data_out(19 downto 15) <= data_in(19 downto 15) when (rvc_t = '0') else --
            RVE Instr
92                              (others => imm_rvc(11)) when (rvc_fctop = "00101"
                                    OR rvc_fctop = "10101") else -- JAL
93                              imm_rvc(7 downto 3) when (rvc_fctop = "01101" AND
                                    lui_en = '1') else -- LUI
94                              "00000" when (rvc_fctop = "01001") else -- c.LI ->
                                    ADDI
95                              rs1;
96
97      data_out(14 downto 12) <= data_in(14 downto 12) when (rvc_t = '0') else --
            RVE Instr
98                              "001" when (rvc_fctop = "11101" OR rvc_fctop = "
                                    00010") else -- BNEZ / SLLI
99                              "010" when (opc /= "01" AND (fct3 = "110" OR fct3 =
                                     "010")) else -- LW/LWSP/SW/SWSP
100                             "100" when (rvc_fctop = "10001" AND fct2 = "11" AND
```

```
                                fct1 = "01") else -- XOR
101                            "101" when (rvc_fctop = "10001" AND data_in(11) =
                                   '0') else -- SRLI / SRAI
102                            "110" when (rvc_fctop = "10001" AND fct2 = "11" AND
                                    fct1 = "10") else -- OR
103                            "111" when (rvc_fctop = "10001" AND (fct2 = "10" OR
                                    fct2 = fct1)) else -- ANDI / AND
104                            (others => imm_rvc(11)) when (rvc_fctop = "00101"
                                   OR rvc_fctop = "10101") else -- JAL
105                            imm_rvc(2 downto 0) when (rvc_fctop = "01101" AND
                                   lui_en = '1') else -- LUI
106                            "000";
107
108    data_out(11 downto 8) <= data_in(11 downto 8) when (rvc_t = '0') else -- RVE
           Instr
109                            imm_rvc(4 downto 1) when (fct3(2 downto 1) = "11")
                                   else -- BEQZ/BNEZ e SW
110                            rd(4 downto 1);
111
112    data_out(7) <= data_in(7) when (rvc_t = '0') else -- RVE Instr
113                   imm_rvc(11) when (fct3(2 downto 1) = "11" AND opc = "01" )
                          else -- BEQZ/BNEZ
114                   imm_rvc(0) when (fct3(2 downto 1) = "11") else -- SW
115                   rd(0);
116
117    data_out(6 downto 2) <= data_in(6 downto 2) when (rvc_t = '0') else -- RVE
           Instr
118                           "01000" when (fct3(2 downto 1) = "11" AND opc(0) = '0')
                                else -- c.SW / c.SWSP -> SW
119                           "00000" when (fct3 = "010" AND opc(0) = '0') else -- c.
                                LW / c.LWSP -> LW
120                           "01101" when (fct3 = "011" AND lui_en = '1') else -- c.
                                LUI -> LUI
121                           "11000" when (fct3(2 downto 1) = "11" AND opc(0) = '1')
                                else -- c.BEQZ / BNEZ -> BEQ / BNE
122                           "11011" when (fct3 = "001" OR fct3 = "101") else -- c.J
                                / c.JAL -> JAL
123                           "11001" when (rvc_fctop = "10010" AND c_jr = '1') else
                                -- c.JR / c.JALR -> JALR
124                           "01100" when ((rvc_fctop = "10010" AND c_jr = '0') OR (
                                rvc_fctop = "10001" AND fct2 = "11")) else -- c.ADD
```

```
                                /c.MV/c.SUB/c.AND/c.OR/c.XOR -> ADD/SUB/AND/OR/XOR
125                            "00100"; -- c.SLLI/c.SRLI/c.SRAI/c.ANDI/c.ADDI/c.LI/c.
                                   ADDI16SP/c.ADDI4SPN -> SLLI/SRLI/SRAI/ANDI/ADDI
126
127     data_out(1 downto 0) <= data_in(1 downto 0);
128     ---------------------------
129     -- rs1, rs2 e rd
130
131     rs2(4) <= '0'; -- base E
132     rs2(3 downto 0) <= "0000" when (fct3(2 downto 1) = "11" AND opc = "01") else
            -- BEQZ/BNEZ
133                        data_in(5 downto 2) when (opc = "10") else -- Q2
134                        '1'&data_in(4 downto 2); -- Q0 e Q1
135
136     rs1(4) <= '0'; -- base E
137     rs1(3 downto 0) <= "0010" when (rvc_fctop = "00000" OR rvc_fctop = "01010"
            OR rvc_fctop = "11010") else -- c.ADDI4SPN, c.LWSP e c.SWSP
138                        "0000" when (rvc_fctop = "10010" AND c_jr = '0' AND
                               data_in(12) = '0') else -- c.MV
139                        '1'&data_in(9 downto 7) when (opc = "00" OR (rvc_fctop(4)
                               = '1' AND opc = "01")) else -- Q0 e parte de Q1
140                        data_in(10 downto 7); -- when (opc = "01" OR opc = "10");
                               -- Q1 e Q2, restante
141
142     rd(4) <= '0'; --base E
143     rd(3 downto 0) <= '1'&data_in(4 downto 2) when (opc = "00") else -- Q0
144                        '1'&data_in(9 downto 7) when (rvc_fctop = "10001") else --
                               Q1, instr. ALU MISC
145                        "0001" when (rvc_fctop = "00101" OR (rvc_fctop = "10010"
                               AND c_jalr = '1')) else -- c.JAL e c.JALR
146                        "0000" when (rvc_fctop = "10101" OR (rvc_fctop = "10010"
                               AND c_jr = '1')) else --c.J e c.JR
147                        data_in(10 downto 7); -- when (opc = "01" OR "10") -- Q1 e
                               Q2, restante
148
149 ----------------------------------------
150 -- Decodificação do imediato
151 ----------------------------------------
152     sig_or_zero <= '1' when (opc = "01") else --sig ext apenas no Q1
153                    '0';
154
```

```
155   mux_imm_zero <= "01" when (rvc_fctop = "01010") else -- LWSP
156           "10" when (rvc_fctop = "11010") else -- SWSP
157           "11" when (rvc_fctop(4 downto 2) = "000") else -- ADDI4SPN
158           "00";          -- LW, SW
159
160   mux_imm_sig <= "01" when (rvc_fctop(4 downto 2) = "011" AND lui_en = '0')
          else -- ADDISP16
161           "10" when (rvc_fctop(4 downto 2) = "101" OR rvc_fctop(4 downto 2) = "
                 001") else -- J, JAL
162           "11" when (fct3(2 downto 1) = "11") else -- BEQZ, BNEZ
163           "00";
164
165   imm_rvc <= (others => '0') when (c_jr = '1' AND rvc_fctop = "10010") else
166              imm_zeroext when (sig_or_zero = '0') else
167              imm_sigext; -- when (sig_or_zero = '1');
168
169   shamt <= data_in(6 downto 2); -- instr c.SLLI esta em um quadrante (opcode)
          separado de c.SRLI/c.SRAI
170   imm_field <= data_in(12 downto 2);
171
172   ----------------------------------------
173   -- Imediato com extensão de sinal, RVC
174   ----------------------------------------
175   imm_sigext(11) <= imm_field(12); -- sinal extendido MSb sempre no [12]
176
177 imm_sigext(10) <= imm_field(8) when (mux_imm_sig = "10") else -- jump
178           imm_field(12); -- sinal extendido do MSb sempre no [12]
179
180 imm_sigext(9)  <= imm_field(10)when (mux_imm_sig = "10") else -- jump
181           imm_field(12); -- sinal extendido do MSb sempre no [12]
182
183 imm_sigext(8)  <= imm_field(9) when (mux_imm_sig = "10") else -- jump
184           imm_field(4) when (mux_imm_sig = "01") else -- addi16sp
185             imm_field(12); -- sinal extendido do MSb sempre no [12]
186
187   imm_sigext(7)  <= imm_field(6) when (mux_imm_sig(1) = '1') else -- jump e
          branch
188              imm_field(3) when (mux_imm_sig(0) = '1') else -- addi16sp
189              imm_field(12); -- sinal extendido do MSb sempre no [12]
190
191   imm_sigext(6)  <= imm_field(5) when (mux_imm_sig(0) = '1') else -- addi16sp
```

```
      e branch
192             imm_field(7) when (mux_imm_sig(1) = '1') else -- jump
193             imm_field(12); -- sinal extendido do MSb sempre no [12]
194
195   imm_sigext(5)  <= imm_field(12) when (mux_imm_sig = "00") else
196             imm_field(2); -- addi16sp, branch e jump
197
198 imm_sigext(4)  <= imm_field(6) when (mux_imm_sig(1) = '0') else
199             imm_field(11);  -- branch e jump
200
201 imm_sigext(3) <= imm_field(10)  when (mux_imm_sig = "11") else -- branch
202             '0'    when (mux_imm_sig = "01") else -- addi16sp
203             imm_field(5);
204
205 imm_sigext(2 downto 1) <= "00" when (mux_imm_sig = "01") else -- "00" quando
    addi16sp
206                 imm_field(4 downto 3);
207
208 imm_sigext(0) <= imm_field(2) when (mux_imm_sig = "00") else '0'; -- '0'
    quando addi16sp, branch ou jump
209
210   ----------------------------------------
211   -- Imediato com extensão de 0s
212   ----------------------------------------
213   imm_zeroext(11 downto 10) <= (others => '0'); -- extenso com 0s
214
215   imm_zeroext(9 downto 8) <= imm_field(10 downto 9) when (mux_imm_zero = "11")
        else -- addi4spn
216                           "00";
217
218 imm_zeroext(7) <= imm_field(8) when (mux_imm_zero(1) = '1') else-- swsp e
    addi4spn
219          imm_field(3) when (mux_imm_zero(0) = '1') else-- lwswp
220          '0';
221
222 imm_zeroext(6) <= imm_field(7) when (mux_imm_zero(1) = '1') else-- swsp e
    addi4spn
223           imm_field(2) when (mux_imm_zero(0) = '1') else-- lwsp
224                imm_field(5); -- lw e sw
225
226 imm_zeroext(5) <= imm_field(12);
```

```
227
228   imm_zeroext(4) <= imm_field(6) when (mux_imm_zero = "01") else -- lwsp
229             imm_field(11); -- lw, sw, addi4spn e swsp
230
231   imm_zeroext(3) <= imm_field(5) when (mux_imm_zero(0) = '1') else-- addi4spn,
          lwsp
232             imm_field(10); -- lw, sw, swsp
233
234   imm_zeroext(2) <= imm_field(4) when (mux_imm_zero = "01") else -- lwsp
235              imm_field(9) when (mux_imm_zero = "10") else -- swsp
236              imm_field(6); -- addi4spn, lw e sw
237
238   imm_zeroext(1 downto 0) <= "00";
239
240 end architecture;
```

APÊNDICE D – DESCRIÇÃO VHDL - DISPOSITIVOS E/S

D.1 – IO_PKG.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  package IO_pkg is
5  -------------------------------------
6  ---------- CONSTANTES
7      constant n : integer := 32;
8      constant n_addrIO : integer := 16;
9
10 -------------------------------------
11 ---------- COMPONENTES
12     COMPONENT UART_control
13         generic(nUART : integer := 8);
14         port(clk : in std_logic;
15             rst  : in std_logic;
16             we_l : in std_logic;
17             cs_UART : in std_logic;
18             RxD  : in std_logic;
19             TxD  : out std_logic;
20             addr_UART : in std_logic_vector(n_addrIO-1 downto 0);
21             data_UART : inout std_logic_vector(n-1 downto 0));
22     end COMPONENT UART_control;
23
24     COMPONENT Timer0
25         port(rst : in std_logic;
26             clk : in std_logic;
27             we_l : in std_logic;
28             cs_T0 : in std_logic;
29             data_in : in std_logic_vector(n-1 downto 0);
30             data_T0 : out std_logic_vector(n-1 downto 0));
31     END COMPONENT Timer0;
32
33 end package IO_pkg;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.IO_pkg.all;
5
6  -- @file IO.vhd
7  -- @author Kevin Morais (moraiskv@gmail)
8  -- @date 2018
9  -- @version 1
10 -- Bloco IO, decodifica e controla o uso dos
11 -- dispositivos IO, UART e Timer0. A UART possui
12 -- ainda um bloco interno próprio para controle, UART_control.vhd.
13 entity IO is
14     port(rst : in std_logic;
15         clk : in std_logic;
16         csIO : in std_logic;
17         we_l : in std_logic;
18         addr : in std_logic_vector(n_addrIO-1 downto 0);
19         dataIO : inout std_logic_vector(n-1 downto 0);
20         ---- conexões da UART
21         RxD : in std_logic;
22         TxD : out std_logic);
23 end entity IO;
24
25 architecture behavior of IO is
26
27     constant LSB_T0 : std_logic_vector(15 downto 0) := x"0008";
28
29     signal cs_UART : std_logic;
30     signal cs_T0 : std_logic;
31
32     signal data_UART: std_logic_vector(n-1 downto 0); -- dado da UART
33     signal data_T0 : std_logic_vector(n-1 downto 0); -- dado do Timer0
34     signal addr_BUS : std_logic_vector(n_addrIO-1 downto 0);-- barramento de end
           . interno do bloco IO
35
36 begin
37
38     addr_BUS <= addr;
```

```
39
40 -- Entrada\Saída de dados
41     data_UART <= dataIO;
42
43     dataIO <= data_T0 when (cs_T0 = '0' AND we_l = '1') else
44                (others => 'Z');
45
46 -- Controle da UART, realiza internamente a comparação com os 16msb de addr_BUS
47     cs_UART <= NOT(csIO); --a UART considera ativo '1'
48
49 -- Controle do Timer0
50     cs_T0 <= '1' when (rst = '1') else
51              '0' when (addr_BUS = LSB_T0 AND csIO = '0') else -- leitura de T0
52              '1';
53
54 -- port map
55   -- instanciação da UART
56     UART_control_i: UART_control port map
57     (clk => clk ,
58     rst => rst ,
59     we_l => we_l ,
60     RxD => RxD ,
61     TxD => TxD ,
62     cs_UART => cs_UART ,
63     addr_UART => addr_BUS ,
64     data_UART => data_UART);
65
66   -- instanciao do Timer0
67   Timer0_i: Timer0 port map
68   (rst => rst ,
69     clk => clk ,
70     we_l => we_l ,
71     cs_T0 => cs_T0 ,
72     data_in => dataIO ,
73     data_T0 => data_T0);
74
75 end architecture behavior;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.IO_pkg.all;
5
6  -- @file Timer0.vhd
7  -- @author Kevin Morais (moraisku@gmail)
8  -- @date 2017
9  -- @version 1
10 -- Temporizador digital que trabalha na mesma frequéncia do
11 -- processador. Possuí um registrador endereçado na memória
12 -- mapeada, x"FFFF_0008".
13 entity Timer0 is
14   port( rst  : in std_logic;        -- sinal de reset do sistema
15       clk : in std_logic;        -- sinal de clk do sistema
16       we_l : in std_logic;        -- sinal de permissão de escrita no T0
17       cs_T0 : in std_logic;        -- sinal de habilitação do T0
18       data_in : in std_logic_vector(n-1 downto 0); -- sinal de entrada do T0
19       data_T0 : out std_logic_vector(n-1 downto 0)); -- sinal de saída do T0
20 end entity Timer0;
21
22 architecture behavior of Timer0 is
23   -- regs. do contador de 32 bits do Timer0, leitura e escrita,
24   -- endereço x"FFFF_0008" do mapeamento na memória dos disp. E/S
25   signal R_T0 : std_logic_vector(n-1 downto 0);
26
27 begin
28   data_T0 <=  R_T0;
29
30   count_i : process(clk , we_l , cs_T0)
31   begin
32     if (rising_edge(clk)) then
33       if (rst = '1') then
34         R_T0 <= (others => '0');
35
36       elsif (we_l = '0' AND cs_T0 = '0') then
37         R_T0 <= data_in;
38
39       else
40         R_T0 <= R_T0 + 1;
41
```

```
42      end if;
43    end if;
44  end process;
45
46 end architecture behavior;
```

D.4 – UART.VHD

```
1  ------------------------------------------------------------------------------
2  -- Title : UART
3  -- Project :
4  ------------------------------------------------------------------------------
5  -- File : UART.vhd
6  -- Author : Giovani Baratto <Giovani.Baratto@ufsm.br>
7  -- Company : UFSM - CT - DELC
8  -- Created : 2017-04-24
9  -- Last update: 2017-06-18
10 -- Platform :
11 -- Standard : VHDL'93/02
12 ------------------------------------------------------------------------------
13 -- Description:
14 ------------------------------------------------------------------------------
15 -- Copyright (c) 2017
16 ------------------------------------------------------------------------------
17 -- Revisions :
18 -- Date Version Author Description
19 -- 2017-04-24 0.1 gfbaratto Created
20 ------------------------------------------------------------------------------
21
22 ------------------------------------------------------------------------------
23 library ieee;
24 use ieee.std_logic_1164.all;
25 use ieee.numeric_std.all;
26 use work.UART_pkg.all;
27 ------------------------------------------------------------------------------
28
29 ------------------------------------------------------------------------------
30 --! @brief sistema de transmissão e recepção UART
31 --! @details Este subsistema UART, transmite e recebe quadros 8N1, com 10 bits.
      A taxa de
```

```
32 --! transmissão édada por UART_clk.
33 --! @author Giovani Baratto (Giovani.Baratto@ufsm.br)
34 --! @version 0.1
35 --! @date 2017
36 --! @todo receber e transmitir quadros com outros formatos
37 --! @image latex block_diagram_uart.eps "Diagrama de bloco da UART" width=\
       textwidth
38 --! @image html block_diagram_uart.png "Diagrama de bloco da UART" width=800
39 entity UART is
40   generic(n_bits : positive := 16); --! número de bits do divisor
41   port(
42     -- transmissor ports
43     UART_Tx_data_in : in std_logic_vector(7 downto 0); --! dado (byte) a ser
         transmitido pela UART
44     UART_Tx_data_out : out std_logic;          --! saída serial do dado a ser
         transmitido
45     UART_Tx_ready : out std_logic;          --! '1' sinaliza que não existe byte
          a ser transmitido
46     UART_Tx_write : in std_logic;          --! '1' (em um ciclo de relógio)
         envia o dado UART_Tx_data_in serialmente.
47     -- receptor ports
48     UART_RX_data_in : in std_logic;          --! dado serial recebido pela UART
49     UART_RX_data_out : out std_logic_vector(7 downto 0); --! dado recebido pela
         UART
50     UART_RX_new_data : out std_logic;          --! '1' sinaliza que um novo dado
          (byte) foi recebido
51     UART_RX_read : in std_logic;          --! '1' indica que o dado recebido foi
          lido: um novo dado pode ser recebido
52     -- UART clock, system clock and reset
53     divisor : in std_logic_vector((n_bits-1) downto 0); --! divide a frequéncia
         de relógio do sistema.
54     clk : in std_logic;          --! relógio do sistema
55     rst : in std_logic          --! reset do sistema
56     );
57 end entity UART;
58 --------------------------------------------------------------------------------
59
60 --------------------------------------------------------------------------------
61 architecture simple of UART is
62   signal UART_clk_16 : std_logic;
63   signal UART_clk : std_logic;
```

```
64  begin
65
66    --! @brief instanciação do susbsistema gerador da taxa de transmissão
67    UART_baud_rate_generator_1 : UART_baud_rate_generator
68      generic map (
69        n_bits => n_bits)
70      port map (
71        divisor => divisor,
72        UART_clk_16 => UART_clk_16,
73        UART_clk => UART_clk,
74        clk => clk,
75        rst => rst);
76
77    --! @brief instanciação do subsistema para a transmissão de um dado(byte),
          serialmente
78    UART_Tx_1 : UART_Tx
79      port map (
80        UART_Tx_data_in => UART_Tx_data_in,
81        UART_Tx_data_out => UART_Tx_data_out,
82        UART_Tx_ready => UART_Tx_ready,
83        UART_Tx_write => UART_Tx_write,
84        UART_clk => UART_clk,
85        clk => clk,
86        rst => rst);
87
88    --! @brief instanciação do subsistema para a recepção serial.
89    UART_RX_1 : UART_RX
90      port map (
91        UART_RX_data_in => UART_RX_data_in,
92        UART_RX_data_out => UART_RX_data_out,
93        UART_RX_new_data => UART_RX_new_data,
94        UART_RX_read => UART_RX_read,
95        UART_clk_16 => UART_clk_16,
96        rst => rst,
97        clk => clk);
98
99  end architecture simple;
100 --------------------------------------------------------------------------------
```

```
1  ----------------------------------------------------------------------------
2  -- Title : UART baud rate generator
3  -- Project : Aulas VHDL
4  ----------------------------------------------------------------------------
5  -- File : UART_clock_generator.vhd
6  -- Author : Giovani Baratto <Giovani.Baratto@ufsm.br>
7  -- Company : UFSM - CT - DELC
8  -- Created : 2017-04-24
9  -- Last update: 2017-06-18
10 -- Platform :
11 -- Standard : VHDL'93/02
12 ----------------------------------------------------------------------------
13 -- Description: baud rate generator to UART
14 ----------------------------------------------------------------------------
15 -- Copyright (c) 2017
16 ----------------------------------------------------------------------------
17 -- Revisions :
18 -- Date Version Author Description
19 -- 2017-04-24 0.1 gfbaratto Created
20 ----------------------------------------------------------------------------
21
22 ----------------------------------------------------------------------------
23 library ieee;
24 use ieee.std_logic_1164.all;
25 use ieee.numeric_std.all;
26 ----------------------------------------------------------------------------
27
28 ----------------------------------------------------------------------------
29 --! @brief Descrição em VHDL da entidade do gerador de baud rate da USART
30 --! @details Descrição em VHDL de um gerador de baud rate da USART. Este éum
31 --! circuito divisor de frequéncia.
32 --! Cada borda de subida do pulso UART_clk, marca um bit de um quadro da
       comunicação.
33 --! A frequéncia do sinal USART_clk édada por:
34 --! \f[ f_{USART\_clk} = \frac{f_{clk}}{16 \cdot (divisor+1)} \f].
35 --! Na saída UART_clk_16 a frequéncia é16 vezes maior e dada pela sequinte equaç
       ão:
36 --! \f[ f_{USART\_clk\_16} = \frac{f_{clk}}{divisor+1} \f]
37 --! @author Giovani Baratto (Giovani.Baratto@ufsm.br)
38 --! @version 0.1
39 --! @date 2017
```

```
40 --! @todo transmitir quadros com outros formatos
41 --! @image latex "block_diagram_uart_baud_rate_generator.eps" "Diagrama de bloco
       do gerador de baud da UART" width=10cm
42 --! @image html "block_diagram_uart_baud_rate_generator.png" "Diagrama de bloco
      do gerador de baud da UART" width=800
43 entity UART_baud_rate_generator is
44   generic(n_bits : positive := 16); --! número de bits do divisor
45   port(divisor : in std_logic_vector((n_bits-1) downto 0); --! divide a frequé
       ncia de clk do sistema
46        UART_clk_16 : out std_logic; --! 16 * relógio da UART
47        UART_clk : out std_logic; --! relógio da UART
48        clk : in std_logic; --! relógio do sistema
49        rst : in std_logic); --! reset assínncrono do sistema
50 end entity UART_baud_rate_generator;
51 --------------------------------------------------------------------------------
52
53 --------------------------------------------------------------------------------
54 architecture simple of UART_baud_rate_generator is
55   signal counter : unsigned(divisor'range); --! contador para realizar a divisão
       da frequéncia de relógio por 16 * (divisor+1)
56   signal counter_16 : unsigned(3 downto 0);
57   signal last_count : std_logic; --! '1' indica que counter está no último valor
       de contagem
58 begin
59
60   last_count <= '1' when std_logic_vector(counter) = divisor else '0';
61   UART_clk_16 <= last_count;
62   UART_clk <= '1' when last_count = '1' and counter_16 = "1111" else '0';
63
64   --! @brief Gera a taxa de transmissão de dados.
65   --! @details Se rst = '1' todos os contadores do divisor são zerados.
66   --! A cada divisor+1 pulsos do sinal de relógio do sistema, égerado um
67   --! pulso na saída UART_clk_16. A cada 16 pulsos de UART_clk_16 ou a cada
68   --! 16 * divisor pulsos do sinal de relógio do sistema égerado um pulso
69   --! na saída UART_clk.
70   baud_rate_p : process(clk, rst)
71   begin
72     if (rst = '1') then -- se rst for igual a '1' zeramos o contador do divisor
        de frequéncia
73       counter <= (others => '0');
74       counter_16 <= (others => '0');
```

```
75     elsif (rising_edge(clk)) then
76       if (last_count = '1') then
77         counter <= (others => '0');
78         counter_16 <= counter_16 + 1;
79       else
80         counter <= counter + 1;
81       end if;
82     end if;
83   end process baud_rate_p;
84
85 end architecture simple;
86 --------------------------------------------------------------------------------
```

D.6 – UART_CONTROL.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.IO_pkg.all;
4
5  -- @file UART_control.vhd
6  -- @author Kevin Morais (moraiskv@gmail)
7  -- @date 2017
8  -- @version 1
9  -- Controle do Dispositivos IO UART.
10 entity UART_control is
11   generic (nUART  : integer := 8); -- tamanho da palavra de dados enviada ou
         recebida pela UART
12   port (
13      -- sinais conectados ao dispositivo IO
14      RxD  : in std_logic;       -- dado serial recebido
15      TxD  : out std_logic;      -- dado serial transmitido
16      -- sinais conectados ao barramento do processador e da memória
17      clk  : in std_logic;       -- sinal de clock
18      rst  : in std_logic;       -- sinal de reset
19      we_l  : in std_logic;       -- sinal de permissão de escrita, ativo em '0'
20      cs_UART : in std_logic;       -- sinal de habilitação de dispostivos IO,
            conectado a UART
21      addr_UART : in std_logic_vector(n_addrIO-1 downto 0);  -- endereço de
            entrada do controle IO
22      data_UART : inout std_logic_vector(n-1 downto 0));-- dado de entrada ou saí
```

```
        da
23   end entity UART_control;
24
25 architecture behavior of UART_control is
26
27   COMPONENT UART
28      generic(n_bits : positive := 16);
29       port( -- transmissor ports
30         UART_Tx_data_in : in std_logic_vector(7 downto 0);
31         UART_Tx_data_out : out std_logic;
32         UART_Tx_ready : out std_logic;
33         UART_Tx_write : in std_logic;
34         -- receptor ports
35         UART_RX_data_in : in std_logic;
36         UART_RX_data_out : out std_logic_vector(7 downto 0);
37         UART_RX_new_data : out std_logic;
38         UART_RX_read : in std_logic;
39         -- UART clock, system clock and reset
40         divisor : in std_logic_vector((n_bits-1) downto 0);
41         clk : in std_logic;
42         rst : in std_logic);
43   end COMPONENT UART;
44
45   signal UART_Tx_data_in : std_logic_vector(nUART-1 downto 0);
46   signal UART_Tx_ready : std_logic;
47   signal UART_Tx_write : std_logic;
48
49   signal UART_RX_new_data : std_logic;
50   signal UART_RX_read : std_logic;
51   signal UART_RX_data_out : std_logic_vector(nUART-1 downto 0);
52
53   -- constante para decodificar os bits [15:0] do end. de acesso do regs. de
        controle da UART
54   constant MSB_CTRL : std_logic_vector(15 downto 0) := x"0000";
55   -- constante para decodificar os bits [15:0] do end. de acesso do regs. de
        recepção ou
56   -- transmissão de dados da UART, se we_l = '0', realiza escrita de TxD, do
        contrário,
57   -- a leitura de RxD
58   constant MSB_RTxD : std_logic_vector(15 downto 0) := x"0004";
59   -- regs. de controle da UART, localizado no end. x"FFFF_0000", somente para
```

```
    leitura
60  signal CTRL_UART : std_logic_vector(n-1 downto 0) := (others => '0');
61  signal s_CTRL  : std_logic; -- habilita o uso do regs. de controle da UART
62  signal s_RTxD : std_logic; -- habilita o uso do regs. de recepção ou transferé
    ncia de dados da UART, em conjunto com 'we_l'
63
64  signal divisor  : std_logic_vector(15 downto 0); -- divisor do clock
65
66 begin
67  -- baud_rate = clk / (16 * (divisor + 1)
68  -- divisor = (clk / (16 * baud_rate) ) - 1
69  --divisor <= x"00A2"; -- divisor = 162 para se obter um baud_rate = 38343
    (~38400)
70  divisor <= x"0145";  -- divisor = 325 para se obter um baud_rate = 19171
    (~19200)
71
72  -- Decodificação da UART
73  s_CTRL  <= '1' when (addr_UART = MSB_CTRL) else '0';
74  s_RTxD  <= '1' when (addr_UART  = MSB_RTxD) else '0';
75
76  -- Leitura do Registrador de Controle da UART, end. x"FFFF_0000"
77  data_UART <= CTRL_UART when (cs_UART = '1' AND s_CTRL = '1' AND we_l = '1')
    else
78       (others => 'Z');
79
80  -- Escrita do Registrador de Transmissão da UART, end. x"FFFF_0004"
81  -- (cs_UART = '1', s_RTxD = '1'e we_l = '0'), mas apenas se a UART
82  -- estiver pronta para enviar novos dados (UART_Tx_ready = '1')
83  UART_Tx_write  <= '1' when (cs_UART = '1' AND s_RTxD = '1' AND we_l = '0' AND
    UART_Tx_ready = '1') else '0';
84  UART_Tx_data_in <= data_UART(nUART-1 downto 0) when (UART_Tx_write = '1') else
85           (others => '0') when rst = '1';
86
87  -- Leitura do Registrador de Recepção da UART, end. x"FFFF_0004"
88  -- (cs_UART = '1', s_RTxD = '1' e we_l = '1'), mas apenas se a UART
89  -- já estiver com um novo dado disponível (UART_Rx_new_data = '1')
90  UART_Rx_read <= '1' when (cs_UART = '1' AND s_RTxD = '1' AND we_l = '1' AND
    UART_Rx_new_data = '1') else '0';
91  data_UART  <= x"000000"&UART_Rx_data_out when (UART_Rx_read = '1') else
92         (others => 'Z');
93
```

```
94   -- Processo de escrita no Registrador de Controle da UART
95   -- Os bits de [31:4] do registrador não so usados, fixos em '0'.
96   CONTROL_UART: process (rst , clk)
97   begin
98     if (rst = '1') then
99       CTRL_UART <= (others => '0');
100
101    elsif (rising_edge(clk)) then
102      CTRL_UART(3) <= UART_Tx_ready;  -- '1' indica que a UART esta pronta para
             enviar o próximo dado
103      CTRL_UART(2) <= UART_Tx_write;  -- '1' indica que a UART recebeu um novo
             dado que esta pronto para ser transmitido
104      CTRL_UART(1) <= UART_Rx_new_data; -- '1' indica que a UART recebeu um novo
             dado que esta pronto para ser lido
105      CTRL_UART(0) <= UART_Rx_read;  -- '1' indica que a UART esta pronta para
             receber o próximo dado
106
107    end if;
108  end process;
109
110  -- instanciação da uart
111  int_serial: UART port map
112  (clk => clk ,
113  rst => rst ,
114  divisor => divisor ,
115  UART_Tx_data_out => TxD ,  -- conectado diretamente a saída TxD do IO_control
116  UART_Tx_data_in => UART_Tx_data_in ,
117  UART_Tx_ready => UART_Tx_ready ,
118  UART_Tx_write => UART_TX_write ,
119  UART_Rx_data_in => RxD , -- conectado diretamente a entrada RxD do IO_control
120  UART_Rx_data_out => UART_Rx_data_out ,
121  UART_Rx_new_data => UART_Rx_new_data ,
122  UART_Rx_read => UART_Rx_read);
123
124 end architecture behavior;
```

D.7 – UART_PKG.VHD

```
1  ----------------------------------------------------------------------------
2  -- Title : Pacote do projeto da UART
```

```
3  -- Project : UART
4  -------------------------------------------------------------------------------
5  -- File : UART_pkg.vhd
6  -- Author : Giovani Baratto <Giovani.Baratto@ufsm.br>
7  -- Company :
8  -- Created : 2017-06-18
9  -- Last update: 2017-06-18
10 -- Platform :
11 -- Standard : VHDL'93/02
12 -------------------------------------------------------------------------------
13 -- Description: Pacote com os procedimento, funções, constantes e componentes
14 -- usados na descrição de uma UART
15 -------------------------------------------------------------------------------
16 -- Copyright (c) 2017
17 -------------------------------------------------------------------------------
18 -- Revisions :
19 -- Date Version Author Description
20 -- 2017-06-18 0.1 Giovani Baratto Created
21 -------------------------------------------------------------------------------
22 -------------------------------------------------------------------------------
23 library ieee;
24 use ieee.std_logic_1164.all;
25 -------------------------------------------------------------------------------
26
27 -------------------------------------------------------------------------------
28 --! brief pacote com os componentes usados na UART
29 package UART_pkg is
30
31   --! @brief declaração do componente para a geração da taxa de transmissão
32   component UART_baud_rate_generator is
33     generic (
34       n_bits : positive);
35     port (
36       divisor : in std_logic_vector((n_bits-1) downto 0);
37       UART_clk_16 : out std_logic;
38       UART_clk : out std_logic;
39       clk : in std_logic;
40       rst : in std_logic);
41   end component UART_baud_rate_generator;
42
43   --! @brief declaração do componente para transmissão de dados (bytes) da UART
```

```
44  component UART_TX is
45    port (
46      UART_TX_data_in : in std_logic_vector(7 downto 0);
47      UART_TX_data_out : out std_logic;
48      UART_TX_ready : out std_logic;
49      UART_TX_write : in std_logic;
50      UART_clk : in std_logic;
51      clk : in std_logic;
52      rst : in std_logic);
53  end component UART_Tx;
54
55  --! @brief declaração do componente para a recepção dos dados (bytes) da UART
56  component UART_RX is
57    port (
58      UART_RX_data_in : in std_logic;
59      UART_RX_data_out : out std_logic_vector(7 downto 0);
60      UART_RX_new_data : out std_logic;
61      UART_RX_read : in std_logic;
62      UART_clk_16 : in std_logic;
63      rst : in std_logic;
64      clk : in std_logic);
65  end component UART_RX;
66
67  --! @brief declaração do componente UART
68  component UART is
69    generic (
70      n_bits : positive);
71    port (
72      UART_Tx_data_in : in std_logic_vector(7 downto 0);
73      UART_Tx_data_out : out std_logic;
74      UART_Tx_ready : out std_logic;
75      UART_Tx_write : in std_logic;
76      UART_RX_data_in : in std_logic;
77      UART_RX_data_out : out std_logic_vector(7 downto 0);
78      UART_RX_new_data : out std_logic;
79      UART_RX_read : in std_logic;
80      divisor : in std_logic_vector((n_bits-1) downto 0);
81      clk : in std_logic;
82      rst : in std_logic);
83  end component UART;
84
```

```
85 end package UART_pkg;
86 ------------------------------------------------------------------------------
```

D.8 – UART_RX.VHD

```
1  ------------------------------------------------------------------------------
2  -- Title : UART Receptor
3  -- Project :
4  ------------------------------------------------------------------------------
5  -- File : UART_RX.vhd
6  -- Author : Giovani Baratto <Giovani.Baratto@ufsm.br>
7  -- Company : UFSM - CT - DELC
8  -- Created : 2017-04-26
9  -- Last update: 2017-06-18
10 -- Platform :
11 -- Standard : VHDL'93/02
12 ------------------------------------------------------------------------------
13 -- Description: UART Receptor description
14 ------------------------------------------------------------------------------
15 -- Copyright (c) 2017
16 ------------------------------------------------------------------------------
17 -- Revisions :
18 -- Date Version Author Description
19 -- 2017-04-26 0.1 gfbaratto Created
20 ------------------------------------------------------------------------------
21
22 ------------------------------------------------------------------------------
23 library ieee;
24 use ieee.std_logic_1164.all;
25 use ieee.numeric_std.all;
26 ------------------------------------------------------------------------------
27
28 ------------------------------------------------------------------------------
29 --! @brief subsistema de recepo da UART
30 --! @details Este subsistema UART, recebe um quadro 8N1, com 10 bits. A taxa de
31 --! transmisso e dada por UART_clk.
32 --! @author Giovani Baratto (Giovani.Baratto@ufsm.br)
33 --! @version 0.1
34 --! @date 2017
35 --! @todo receber quadros com outros formatos
```

```
36 --! @image latex block_diagram_uart_rx.eps "Diagrama de bloco do receptor UART"
       width=10cm
37 --! @image html block_diagram_uart_rx.png "Diagrama de bloco do receptor UART"
       width=800
38 entity UART_RX is
39   port(UART_RX_data_in : in std_logic;       --! dados de entrada seriais
40        UART_RX_data_out : out std_logic_vector(7 downto 0); --! dado (byte)
             recebido pela entrada serial UART_RX_data_in
41        UART_RX_new_data : out std_logic;       --! '1' indica que um novo dado
             foi recebido
42        UART_RX_read : in std_logic;       --! '1' permite que novos dados sejam
             recebidos pela entrada serial
43        UART_clk_16 : in std_logic;       --! 16 * relgio da UART
44        rst : in std_logic;       --! sinal de reset do sistema
45        clk : in std_logic); --! relgio do sistema
46 end entity UART_RX;
47 --------------------------------------------------------------------------------
48
49 --------------------------------------------------------------------------------
50 architecture simple of UART_RX is
51   signal sample_register : std_logic_vector(2 downto 0);
52   signal data_register : std_logic_vector(7 downto 0);
53   signal edge_neg : std_logic;
54   signal samples_equal : std_logic;
55   signal counter : unsigned(3 downto 0);
56
57   signal new_data : std_logic;
58   signal sample_counter : unsigned(3 downto 0);
59   signal data_counter : unsigned(3 downto 0);
60   signal sample_counter_clear : std_logic;
61   signal data_counter_clear : std_logic;
62   signal enable : std_logic;
63   signal enable_clear : std_logic;
64   signal sample : std_logic;
65   signal transfer : std_logic;
66
67   signal tf_en : std_logic; -- desabilita transfer após o primeiro pulso,
       evitando erros de temporização na recepção
68
69 begin
70
```

```
71  enable_clear <= data_counter(3) and data_counter(1);
72  sample <= '1' when sample_counter = "0111" and UART_clk_16 = '1' else '0';
73  transfer <= '1' when data_counter = "1001" and UART_clk_16 = '1' and tf_en =
        '0' else '0'; -- teste
74
75
76  UART_RX_new_data <= new_data;
77
78  enable_p : process(clk, rst)
79  begin
80    if (rst = '1') then
81      enable <= '0';
82    elsif(rising_edge(clk)) then
83      if(enable = '0') then
84        enable <= not (UART_RX_data_in or new_data);
85      else
86        enable <= not enable_clear;
87      end if;
88    end if;
89  end process enable_p;
90
91  sample_counter_p : process(clk, rst)
92  begin
93    if (rst = '1') then
94      sample_counter <= (others => '0');
95    elsif(rising_edge(clk)) then
96      if(enable = '1') then
97        if(UART_clk_16 = '1') then
98          sample_counter <= sample_counter + 1;
99        else
100         sample_counter <= sample_counter;
101       end if;
102     else
103       sample_counter <= (others => '0');
104     end if;
105   end if;
106 end process sample_counter_p;
107
108
109 data_counter_p : process(clk, rst)
110 begin
```

```
111   if(rst = '1') then
112     data_counter <= (others => '0');
113   elsif(rising_edge(clk)) then
114     if(enable = '1') then
115       if(sample = '1') then
116         data_counter <= data_counter + 1;
117       end if;
118     else
119       data_counter <= (others => '0');
120     end if;
121   end if;
122 end process data_counter_p;
123 
124 data_register_p : process(clk, rst)
125 begin
126   if(rst = '1') then
127     data_register <= (others => '0');
128   elsif(rising_edge(clk)) then
129     if(sample = '1') then
130       data_register <= UART_RX_data_in & data_register(7 downto 1);
131     end if;
132   end if;
133 end process data_register_p;
134 
135 data_out : process(clk, rst)
136 begin
137   if(rst = '1') then
138     UART_RX_data_out <= (others => '0');
139   elsif(rising_edge(clk)) then
140     if(transfer = '1') then
141       UART_RX_data_out <= data_register;
142     end if;
143   end if;
144 end process data_out;
145 
146 process(clk, rst)
147 begin
148   if (rst = '1') then
149     new_data <= '0';
150  elsif(rising_edge(clk)) then
151     if(transfer = '1') then
```

```
152         new_data <= '1';
153       elsif(UART_RX_read = '1') then
154         new_data <= '0';
155       end if;
156     end if;
157   end process;
158
159
160  -- teste
161   process(clk , rst)
162   begin
163   if (rst = '1') then
164     tf_en <= '0';
165   elsif (rising_edge(clk)) then
166     if (transfer = '1') then
167       tf_en <= '1';
168     elsif (enable = '0') then
169       tf_en <= '0';
170     end if;
171    end if;
172   end process;
173 -- teste
174
175 end architecture simple;
176 --------------------------------------------------------------------------------
```

D.9 – UART_TX.VHD

```
1  --------------------------------------------------------------------------------
2  -- Title : UART_TX
3  -- Project : UART
4  --------------------------------------------------------------------------------
5  -- File : UART_TX.vhd
6  -- Author : Giovani Baratto <gfbaratto@UFSM-notebook>
7  -- Company : UFSM - CT - DELC
8  -- Created : 2017-04-26
9  -- Last update: 2017-06-18
10 -- Platform :
11 -- Standard : VHDL'93/02
12 --------------------------------------------------------------------------------
```

```
13 -- Description:
14 ----------------------------------------------------------------------------
15 -- Copyright (c) 2017
16 ----------------------------------------------------------------------------
17 -- Revisions :
18 -- Date Version Author Description
19 -- 2017-04-26 0.1 gfbaratto Created
20 ----------------------------------------------------------------------------
21
22 ----------------------------------------------------------------------------
23 library ieee;
24 use ieee.std_logic_1164.all;
25 ----------------------------------------------------------------------------
26
27 ----------------------------------------------------------------------------
28 --! @brief subsistema de transmissão da UART
29 --! @details Este subsistema UART, transmite um quadro 8N1, com 10 bits. A taxa
       de
30 --! transmissão édada por UART_clk.
31 --! @author Giovani Baratto (Giovani.Baratto@ufsm.br)
32 --! @version 0.1
33 --! @date 2017
34 --! @todo transmitir quadros com outros formatos
35 --! @image latex block_diagram_uart_tx.eps "Diagrama de bloco do transmissor
       UART" width=10cm
36 --! @image html block_diagram_uart_tx.png "Diagrama de bloco do transmissor UART
       " width=800
37 entity UART_TX is
38   port(UART_TX_data_in : in std_logic_vector(7 downto 0); --! vetor com dados de
         entrada
39        UART_TX_data_out : out std_logic; --! saída dos dados de entrada,
             transmitidos serialmente
40        UART_TX_ready : out std_logic; --! se '1', novos dados de entrada são
             aceitos
41        UART_TX_write : in std_logic; --! se '1', envia UART_TX_data_in
42        UART_clk : in std_logic; --! relógio do transmissor UART
43        clk : in std_logic; --! relógio do sistema
44        rst : in std_logic); --! reset assíncrono do transmissor
45 end entity UART_TX;
46 ----------------------------------------------------------------------------
47
```

```
48 ----------------------------------------------------------------------------
49 architecture simple of UART_TX is
50
51   constant bits_in_frame : integer := 10; --! número de bits em um quadro
52   signal sh_register : std_logic_vector(0 to bits_in_frame-1); --! registrador
         de deslocamento
53   signal counter : integer range 0 to bits_in_frame; --! conta o número de bits
         que devem ser enviados
54
55 begin
56
57   UART_Tx_ready <= '1' when counter = 0 else '0'; -- se '1', UART pronta para
         enviar novos dados
58
59   --!
60   UART_TX_p : process(UART_Tx_write, clk, rst)
61   begin
62     if (rst = '1') then      -- se reset = '1'
63       sh_register <= (others => '0');  -- o registrador de deslocamento ézerado
64       UART_Tx_data_out <= '1';      -- a saída écolocada em '1'
65       counter <= 0;          -- o contador de bits a ser enviado ézerado: não
             existem bits para serem enviados
66
67     elsif (rising_edge(clk)) then      -- senão, se temos uma borda de subida do
           sinal de relégio
68       if (counter /= 0) then      -- se existem bits a serem transmitidos
69         if (UART_clk = '1') then      -- se temos um pulso do relógio da UART,
               podemos transmitir novo bit
70           UART_Tx_data_out <= sh_register(bits_in_frame-1);     -- enviamos o pró
                ximo bit do registrador de deslocamento
71           sh_register <= '1' & sh_register(0 to bits_in_frame-2); -- deslocamos o
                registrador de deslocamento
72           counter <= counter - 1;  -- decrementamos o contador: mais um bit foi
                enviado
73         end if;
74
75       elsif (UART_Tx_write = '1') then      -- se counter=0 (sem bits para enviar
           ) e temos uma solicitação de envio de novo byte
76         sh_register <= '1' & UART_Tx_data_in & '0';  -- registramos no
               registrador de deslocamento um quadro para transmissão
77         counter <= bits_in_frame;      -- atualizamos o número de bits que serão
```

```
            transmitidos
      end if;
    end if;
  end process UART_TX_p;

end architecture simple;
--------------------------------------------------------------------------------
```

APÊNDICE E – DESCRIÇÃO VHDL - NÚCLEO MULTICÍCLICO

Possuí dois arquivos de implementação, um para o núcleo apenas, e outro contando com todo o sistema. Dois arquivos VHDL de topo da hierarquia também, para testbench e implementação final.

E.1 – CORE.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.core_pkg.all;
4
5  -- @file core.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2017
8  -- @version 1
9  -- Núcleo RVC Multiciclo.
10 -- Conecta todos os blocos núcleo, realizando as lógicas do Caminho
11 -- de Dados, como os multiplexadores controlados pelos sinais provenientes da
       Unidade de Controle,
12 -- ou então os registradores temporários (A, B, Instrução e ALU).
13 entity core is
14   port(clk : in std_logic; -- sinal de clk
15     rst  : in std_logic; -- sinal de reset, ativo em '1'
16     wait_mem : in std_logic; -- sinal wait_mem, indicando quando um dado está
           pronto
17     we_l : out std_logic; -- write enable low, para memórias e disp. E/S
18     Valid_addr : out std_logic; -- sinal de validade do endereço do barramento
19     addr_CPU : out std_logic_vector(n-1 downto 0); -- saída de endereço do nú
           cleo
20     data_CPU : inout std_logic_vector(n-1 downto 0)); -- saida/entrada de dados
           do núcleo
21 end entity core;
22
23 architecture behavior of core is
24
25 ---------------------------------------------------------------------
26 -- Sinais
```

```
27 ----------------------------------------------------------------------
28   --Sinais de Controle
29   signal PC_W  : std_logic; -- sinal de permissão de escrita no ProgramCounter (
         PC)
30   signal IouD  : std_logic; -- sinal que controla o multiplexador 1 (Mux 1 - M1)
          do Caminho de Dados, define o endereço contido na saída addr_CPU
31   signal Mem_RW : std_logic; -- sinal de permissão de leitura ou escrita em
         componentes externos ao processador, '1' = leitura e '0' = escrita
32   signal Instr_W : std_logic; -- sinal de permissao de escrita no Registrador de
          Instrução
33   signal MemToReg : std_logic_vector(1 downto 0); -- sinal que controla o
         multiplexador 2 (Mux 2 - M2) do Caminho de Dados, define a entrada de dados
          do Reg_File (Banco de Regs.)-
34   signal RegDst_A : std_logic; -- sinal que controla o multiplexador 7 (Mux 7 -
         M7) do Caminho de Dados, define a entrada de endereço do Reg_File
35   signal RegDst_W : std_logic_vector(1 downto 0); -- sinal que controla o
         multiplexador 3 (Mux 3 - M3) do Caminho de Dados, define o endereço de
         escrita do Reg_File
36   signal RegAcess : std_logic; -- sinal que define se o acesso aos registradores
          de Reg_File (Banco de Regs.) ser limitado de x8 a x15 ou no
37   signal PC_W_cond: std_logic; -- sinal de permissão de escrita condicional no
         ProgramCounter (PC)
38   signal PCFonte : std_logic; -- sinal que controla o multiplexador 6 (Mux 6 -
         M6) do Caminho de Dados, define a entrada do ProgramCounter (PC)
39   signal Imm_op : std_logic_vector(3 downto 0); -- sinal que controla os
         multiplexadores do bloco Imm, decodifica o immediate da instrução atual
40   signal ALUoprB : std_logic_vector(1 downto 0); -- sinal que define o
         operando_B da ALU, controla o multiplexador 5 (Mux 5 - M5)
41   signal AluoprA : std_logic_vector(1 downto 0); -- sinal que define o
         operando_A da ALU, controla o multiplexador 4 (Mux 4 - M4)
42   signal ALUop : std_logic_vector(1 downto 0); -- sinal da Unidade de Controle
         conectado como entrada na Subunidade de Controle da ALU, para definir a
         operação da mesma
43   signal RegFile_W: std_logic; -- sinal de permissão de escrita no Reg_File (
         Banco de Regs.)
44
45   -- Sinais de entrada da Unidade de Controle Principal, da Subunidade de
         Controle e
46   -- das entradas e saídas da Subunidade da ALU
47   signal jr_en  : std_logic; -- sinal da subunidade de controle que decodifica a
          instrução c.JR e c.JALR
```

```
48  signal addi16sp_en : std_logic;  -- sinal da subunidade de controle que
        decodifica a instrução c.ADDI16SP
49  signal funct4  : std_logic; -- bit 12 da instrução, utilizado para decodificaç
        ão das instruções c.MV, c.ADD, c.JR e c.JALR, conectado a Unidade de
        Controle
50  signal funct3_op : std_logic_vector(4 downto 0); -- bits [15:13|1:0] da instru
        ção, conectados a Unidade de Controle para decodificação da instrução atual
51  signal ft2_temp  : std_logic;     -- sinal conectado a Unidade de Controle,
        para instruções possuem o mesmo opcode
52  signal ALU_control : std_logic_vector(3 downto 0); -- sinal de saída da
        Subunidade de Controle da ALU, define a operação que a mesma irá realizar
53
54  -- Sinais da unidade Immediate
55  signal imm_sigext : std_logic_vector(n-1 downto 0); -- campo immediate com
        sinal extendido pelo MSb
56  signal imm_zeroext : std_logic_vector(n-1 downto 0); -- campo immediate com
        sinal extendido com 0s
57  signal imm_lui  : std_logic_vector(n-1 downto 0); -- campo immediate para a
        instrução c.LUI
58  signal instr  : std_logic_vector(10 downto 0); -- campo da instrução que contm
         todos os immediates, bits de [12:2] da instrução, conectado a Imm.vhd para
         decodificação
59  signal imm_shift : std_logic_vector(4 downto 0);  -- campo immediate que contm
         o nmero de bits a serem deslocados em instruções de shift, conectado
        diretamente a ALU
60
61  -- Sinais restantes
62  signal PC   : std_logic_vector(n-1 downto 0);  -- valor atual do
        ProgramCounter (PC)
63  signal PC_next : std_logic_vector(n-1 downto 0);  -- valor de entrada do
        ProgramCounter (PC)
64  signal PC_we : std_logic;      -- sinal de habilitação da escrita do
        ProgramCounter (PC)
65  signal address_A: std_logic_vector(3 downto 0);  -- endereço de acesso A para
        leitura do Reg_File (Banco de Regs.), saída do MUX 7
66  signal address_B: std_logic_vector(3 downto 0);  -- endereço de acesso B para
        leitura do Reg_File (Banco de Regs.)
67  signal address_W: std_logic_vector(3 downto 0);  -- endereço de acesso W para
        escrita do Reg_File (Banco de Regs.)
68  signal data_in_RF: std_logic_vector(n-1 downto 0); -- dado de entrada do
        Reg_File (Banco de Regs.)
```

```
69   signal data_out_A: std_logic_vector(n-1 downto 0);  -- dado de saída A do
        Reg_File (Banco de Regs.)
70   signal data_out_B: std_logic_vector(n-1 downto 0);  -- dado de saída B do
        Reg_File (Banco de Regs.)
71   signal operando_A: std_logic_vector(n-1 downto 0);  -- operando A da ALU
72   signal operando_B: std_logic_vector(n-1 downto 0);  -- operando B da ALU
73   signal ALUzero : std_logic;      -- saída de verificação de igualdade a zero
        do resultado da ALU, podendo ser negado na instrução c.BNEZ
74   signal ALU_out : std_logic_vector(n-1 downto 0);  -- saída da ALU, sendo o
        resultado da operação entre operando_A e operando_B
75
76   -- campos da instrução
77   signal funct2 : std_logic_vector(1 downto 0); -- bits [11:10] da instrução,
        utilizados para determinar a operação da ALU, conectados a Subunidade de
        Controle da ALU
78   signal funct1 : std_logic_vector(1 downto 0); -- bits [6:5] da instrução,
        utilizados para determinar a operação da ALU, conectados a Subunidade de
        Controle da ALU
79   signal rs1 : std_logic_vector(3 downto 0); -- endereço de acesso A para
        leitura do Reg_file (Banco de Regs.), provido pela instrução
80   -- Sinais dos Registradores temporários
81   signal RegA : std_logic_vector(n-1 downto 0);  -- registrador que armazena a
        saída A (data_out_A) do Reg_File (Banco de Regs.)
82   signal RegB : std_logic_vector(n-1 downto 0);  -- registrador que armazena a
        saída B (data_out_B) do Reg_File (Banco de Regs.)
83   signal RegData : std_logic_vector(n-1 downto 0); -- registrador que armazena o
         dado lido da Memória ou algum dispositivo IO
84   signal RegALU : std_logic_vector(n-1 downto 0); -- registrador que armazena o
        resultado da operação da ALU
85   signal RegInstr : std_logic_vector(nC-1 downto 0);  -- registrador que
        armazena a instrução em execução
86
87 begin
88
89 -- lógica de habilitação da escrita do ProgramCounter (PC)
90   PC_we <= ((PC_W_cond AND ALUzero) OR PC_W) AND NOT(wait_mem);
91
92 -- Permissão de escrita na Memória de Dados, ativo em nível '0'
93   we_l <= Mem_RW;
94   data_CPU <= RegB when (Mem_RW = '0') else (others => 'Z');
95
```

```
96  -- Sinais de saída do Registrador de Instruções
97    rs1   <= RegInstr(10 downto 7); -- end. do banco de reg., A e B, o A ainda é
          definido
98    address_B  <= RegInstr(5 downto 2); -- após a lógica de rs1 com o sinal
          RegDst_A
99
100   instr  <= RegInstr(12 downto 2); -- campo que contém imediato embaralhado
101   imm_shift <= RegInstr(6 downto 2); -- campo que contém o valor de shift (c.
          SRLI, c.SRAI e c.SLLI)
102
103 -- Sinais conectados a unidade de controle, para decodificação
104   funct3_op <= RegInstr(15 downto 13)&RegInstr(1 downto 0); -- campo principal
          de decodificação
105   funct4  <= RegInstr(12);   -- diferencia instr. com mesmo funct3_op (MV, ADD,
          JR e JALR)
106
107 -- Sinais conectados a subunidade de controle
108   funct2  <= RegInstr(11 downto 10); -- decodifica instr. de operações Reg to
          Imm
109   funct1  <= RegInstr(6 downto 5); -- decodifica instr. de operações Reg to Reg
110
111 -- Multiplexadores
112   -- Mux 1 - Multiplexador 1 - M1, controlado por IouD, escolhe
113   -- o endereço de acesso da memória, entre PC e RegALU
114   addr_CPU <= PC  when IouD = '0' else
115         RegALU when IouD = '1';
116
117   -- Mux 2 - Multiplexador 2 - M2, controlado por MemToReg, escolhe
118   -- o dado a ser escrito no banco de registradores (RegFile)
119   data_in_RF <= RegData  when MemToReg = "00" else
120           RegALU  when MemToReg = "01" else
121           PC    when MemToReg = "10" else
122           imm_lui  when MemToReg = "11";
123
124   -- Mux 3 - Multiplexador 3 - M3, controlado por RegDst_W, escolhe
125   -- o endereço de escrita do RegFile
126   address_W <=  RegInstr(10 downto 7)  when RegDst_W = "00" else
127           --RegInstr(5 downto 2) when RegDst_W = "01" else
128           x"1"     when RegDst_W = "10" else
129           RegInstr(5 downto 2);
130
```

```
131   -- Mux 4 - Multiplexador 4 - M4, controlado por ALUoprA, escolhe
132   -- a entrada A da ALU
133   operando_A <= PC    when ALUoprA = "00" else
134          RegA  when ALUoprA = "01" else
135          x"00000000" when ALUoprA = "10" else
136          RegA;
137
138   -- Mux 5 - Multiplexador 5 - M5, controlado por ALUoprB, escolhe
139   -- a entrada B da ALU
140   operando_B <=  RegB  when ALUoprB = "00" else
141          x"00000002" when ALUoprB = "01" else
142          imm_sigext when ALUoprB = "10" else
143          imm_zeroext when ALUoprB = "11";
144
145   -- Mux 6 - Multiplexador 6 - M6, controlado por PCFonte, escolhe qual
146   -- será o próximo end. de PC, entre a saída da ALU, e o RegALU
147   PC_next <= ALU_out when PCFonte = '0' else
148        RegALU when PCFonte = '1';
149
150   -- Mux 7 - Multiplexador 7 - M7, controlado por RegDst_A, escolhe
151   -- qual sera o endereço de acesso A do banco de regs. entre rs1 e 2
152   address_A <=  rs1  when RegDst_A = '0' else
153          x"2"  when RegDst_A = '1';
154
155 --- Subunidade de Controle
156   addi16sp_en <= '1' when RegInstr(10 downto 7) = "0010" else '0';
157   jr_en  <= '1' when RegInstr(5 downto 2) = "0000" else '0';
158   ft2_temp <= funct2(1) AND funct2(0);
159
160 --- Subunidade de Controle da ALU, define a operação do bloco de acordo com os
        campos funct1 e ALUop
161   ALU_control <= "0000" when ALUop = "11"     else -- ADD , BEQZ (utiliza a soma
        )
162          "1000"  when ALUop = "10"     else -- ADD , BNEZ
163          "0101"  when ALUop = "01"     else -- SLL
164          "0110"  when funct2 = "00"     else -- SRL
165          "0111"  when funct2 = "01"     else -- SRA
166          "0011" when (funct2 = "10" OR funct1 = "11") else  -- AND
167          "0001"  when funct1 = "00"     else -- SUB
168          "0100"  when funct1 = "01"     else -- XOR
169          "0010"  when funct1 = "10"     else -- OR
```

```
170         "0000"; -- ADD
171
172   -- escreve nos regs. temporarios do caminho de dados
173   Reg_Temp: process(rst , clk , wait_mem)
174   begin
175     if (rising_edge(clk)) then
176           if (rst ='1') then
177               RegData <= (others => '0');
178               RegA  <= (others => '0');
179               RegB  <= (others => '0');
180               RegALU  <= (others => '0');
181
182       elsif (wait_mem = '1') then
183         --não faz nada
184
185           else
186               RegData <= data_CPU; -- escreve a saída da memória em RegData
187               RegA  <= data_out_A; -- escreve a saída A de RegFile no RegA
188               RegB  <= data_out_B; -- escreve a saída B de RegFile no RegB
189               RegALU  <= ALU_out;  -- escreve a saída da ALU no RegALU
190
191           end if;
192         end if;
193   end process Reg_Temp;
194
195   -- escreve no regs. de instruções do caminho de dados
196   Reg_Instr : process(rst , clk , Instr_W)
197   begin
198     if (rising_edge(clk)) then
199           if (rst = '1') then
200               RegInstr  <= (others => '0');
201
202       elsif (wait_mem = '1') then
203         --não faz nada
204
205           elsif (Instr_W = '1') then
206               RegInstr <= data_CPU(nC-1 downto 0);
207
208           end if;
209         end if;
210   end process Reg_Instr;
```

```
211
212  -- instanciação do bloco PC
213  ProgramCounter_i: ProgramCounter port map
214  (rst  => rst ,
215  clk  => clk ,
216  PC_we  => PC_we ,
217  PC_next => PC_next ,
218  PC   => PC);
219
220  -- instanciação do bloco RegFile
221  RegFile_i: RegFile port map
222  (rst   => rst ,
223  clk   => clk ,
224  RegFile_W  => RegFile_W ,
225  RegAcess  => RegAcess,
226  RegDst_A => RegDst_A ,
227  address_W  => address_W ,
228  address_A => address_A ,
229  address_B  => address_B ,
230  data_in_RF  => data_in_RF ,
231  data_out_A  => data_out_A ,
232  data_out_B  => data_out_B);
233
234  -- instanciação do bloco ALU
235  ALU_i: ALU  port map
236  (operando_A => operando_A ,
237  operando_B  => operando_B ,
238  ALU_control => ALU_control ,
239  ALU_out  => ALU_out ,
240  ALUzero  => ALUzero ,
241  imm_shift  => imm_shift);
242
243  -- instanciação do bloco Un_Control
244  Un_Control_i: Un_Control port map
245  (rst   => rst ,
246  clk   => clk ,
247  wait_mem  => wait_mem ,
248  funct3_op  => funct3_op ,
249  addi16sp_en => addi16sp_en ,
250  jr_en   => jr_en ,
251  funct4   => funct4 ,
```

```
252   ft2_temp  => ft2_temp ,
253   PC_W   => PC_W ,
254   PCFonte  => PCFonte ,
255   PC_W_cond  => PC_W_cond ,
256   IouD   => IouD ,
257   Mem_RW   => Mem_RW ,
258   Valid_addr => Valid_addr ,
259   Instr_W  => Instr_W ,
260   MemToReg  => MemToReg ,
261   RegDst_A  => RegDst_A ,
262   RegDst_W  => RegDst_W ,
263   RegAcess  => RegAcess ,
264   ALUop   => ALUop ,
265   ALUoprB  => ALUoprB ,
266   ALUoprA  => ALUoprA ,
267   RegFile_W  => RegFile_W ,
268   Imm_op   => Imm_op);
269
270   -- instanciação do bloco Imm
271   Imm_i: Imm port map
272   (imm_sigext => imm_sigext ,
273   imm_zeroext => imm_zeroext ,
274   imm_lui  => imm_lui ,
275   Imm_op   => Imm_op ,
276   instr   => instr);
277
278 end architecture behavior;
```

E.2 – CORE_PKG.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  -- Pacote com os componentes e sinais utilizados no núcleo.
5  package core_pkg is
6
7     constant n : integer := 32; -- tamanho dos dados
8     constant nC: integer := 16; -- tamanho das instruções
9
10 ---------------------------------------------
```

```
11 -- Componentes
12 ---------------------------------------------
13     -- componente Program Counter (PC)
14     COMPONENT ProgramCounter
15         port(clk : in std_logic;
16             rst : in std_logic;
17             PC_we : in std_logic;
18             PC_next : in std_logic_vector(n-1 downto 0);
19             PC : out std_logic_vector(n-1 downto 0));
20     end COMPONENT ProgramCounter;
21 
22     -- componente RegFile
23     COMPONENT RegFile
24         port(clk : in std_logic;
25             rst : in std_logic;
26             RegFile_W : in std_logic;
27             RegAcess : in std_logic;
28             RegDst_A : in std_logic;
29             address_A : in std_logic_vector(3 downto 0);
30             address_B : in std_logic_vector(3 downto 0);
31             address_W : in std_logic_vector(3 downto 0);
32             data_in_RF : in std_logic_vector(n-1 downto 0);
33             data_out_A : out std_logic_vector(n-1 downto 0);
34             data_out_B : out std_logic_vector(n-1 downto 0));
35     end COMPONENT RegFile;
36 
37     -- componente ALU
38     COMPONENT ALU
39         port(operando_A : in std_logic_vector(n-1 downto 0);
40             operando_B : in std_logic_vector(n-1 downto 0);
41             ALU_control : in std_logic_vector(3 downto 0);
42             imm_shift : in std_logic_vector(4 downto 0);
43             ALU_out : out std_logic_vector(n-1 downto 0);
44             ALUzero : out std_logic);
45     end COMPONENT ALU;
46 
47     -- componente Unidade de Controle
48     COMPONENT Un_Control
49         port(clk : in std_logic;
50             rst : in std_logic;
51             wait_mem : in std_logic;
```

```
52          jr_en : in std_logic;
53          funct4 : in std_logic;
54          ft2_temp : in std_logic;
55          addi16sp_en : in std_logic;
56          funct3_op : in std_logic_vector(4 downto 0);
57          Imm_op : out std_logic_vector(3 downto 0);
58          PC_W : out std_logic;
59          PC_W_cond : out std_logic;
60          PCFonte : out std_logic;
61          RegDst_A : out std_logic;
62          RegFile_W : out std_logic;
63          RegAcess : out std_logic;
64          MemToReg : out std_logic_vector(1 downto 0);
65          RegDst_W : out std_logic_vector(1 downto 0);
66          IouD : out std_logic;
67          Instr_W : out std_logic;
68          Mem_RW : out std_logic;
69          Valid_addr : out std_logic;
70          ALUop : out std_logic_vector(1 downto 0);
71          ALUoprB : out std_logic_vector(1 downto 0);
72          ALUoprA : out std_logic_vector(1 downto 0));
73    end COMPONENT Un_Control;
74
75    -- componente Imm
76    COMPONENT Imm
77    port(Imm_op : in std_logic_vector(3 downto 0);
78        instr : in std_logic_vector(12 downto 2);
79        imm_sigext : out std_logic_vector(n-1 downto 0);
80        imm_zeroext : out std_logic_vector(n-1 downto 0);
81        imm_lui : out std_logic_vector(n-1 downto 0));
82    end COMPONENT Imm;
83
84 end package core_pkg;
```

E.3 – ALU.VHD

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 use ieee.numeric_std.all;
```

```
5  use work.core_pkg.all;
6
7  -- @file ALU.vhd
8  -- @author Kevin Morais (moraiskv@gmail)
9  -- @date 2017
10 -- @version 1
11 -- Unidade Lógica e Aritmética, realiza a operação definida pelo sinal de
12 -- controle (ALU_control), sobre dois operandos (opperando_A, operando_B),
13 -- obtendo-se o resultado (ALU_out). Operaçóes de deslocamentos são de
14 -- acordo com o sinal imm_shift. ALUzero indica quando o operando_A possui
15 -- valor 0 para instruçóes c.BEQZ, ou diferente de 9 para BNEZ.
16 entity ALU is
17   PORT(operando_A : in std_logic_vector(n-1 downto 0);
18     operando_B : in std_logic_vector(n-1 downto 0);
19     ALU_control : in std_logic_vector(3 downto 0);
20     imm_shift : in std_logic_vector(4 downto 0);
21     ALU_out  : out std_logic_vector(n-1 downto 0);
22     ALUzero  : out std_logic);
23 end entity ALU;
24
25 architecture behavior of ALU is
26   -- sinal utilizado para comparação do operando_A a 0, negado em instr. c.BNEZ,
27   -- a inversão édefinida pelo MSB do sinal de entrada ALU_control
28   signal ALU_ctrl_temp : std_logic_vector(2 downto 0); -- campo que define a
       operação
29   signal shamt: integer range 0 to 31; -- sinal shamt, imm_shift convertido para
        integer
30   signal zero : std_logic;
31
32 begin
33
34   zero  <= '1' when (operando_A = x"00000000") else '0';
35   ALUzero <= zero XOR ALU_control(3);
36   shamt <= to_integer(unsigned(imm_shift)); -- conversão para utilizar SRA, SRL
       e SLL
37   ALU_ctrl_temp <= ALU_control(2 downto 0);
38
39   ALU_out <=      operando_A + operando_B when ALU_ctrl_temp = "000" else
40                operando_A - operando_B when ALU_ctrl_temp = "001" else
41                operando_A OR operando_B when ALU_ctrl_temp = "010" else
42                operando_A AND operando_B when ALU_ctrl_temp = "011" else
```

```
43              operando_A XOR operando_B when ALU_ctrl_temp = "100" else
44 to_stdlogicvector(to_bitvector(operando_A) SLL shamt)  when ALU_ctrl_temp = "101
       " else
45 to_stdlogicvector(to_bitvector(operando_A) SRL shamt)  when ALU_ctrl_temp = "110
       " else
46 to_stdlogicvector(to_bitvector(operando_A) SRA shamt)  when ALU_ctrl_temp = "111
       ";
47
48 end architecture behavior;
```

E.4 – IMM.VHD

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use work.core_pkg.all;
 4
 5 -- @file Imm.vhd
 6 -- @author Kevin Morais (moraisku@gmail)
 7 -- @date 2017
 8 -- @version 1
 9 -- Decodifica o campo imediato embaralhado ao longo da instrução.
10 -- O MSB (em estenções com sinal), está sempre localizado no bit [12]
11 -- da instrução. O sinal de controle éo Imm_op, sendo os 2 MSB
12 -- responsáveis pelo controle do imediato estendido com sinal, e os
13 -- 2 LSB pela sinal com zeros. A entrada 'instr' contém os campos
14 -- a serem desembaralhados para formar o imediato. Sendo as saídas
15 -- resultantes imm_sigext, imm_zeroext e imm_lui, sendo a ultima
16 -- unicamente para instruções c.LUI.
17 entity Imm is
18   port( Imm_op : in std_logic_vector(3 downto 0);
19     instr  : in std_logic_vector(12 downto 2);
20     imm_sigext : out std_logic_vector(n-1 downto 0);
21     imm_zeroext : out std_logic_vector(n-1 downto 0);
22     imm_lui  : out std_logic_vector(n-1 downto 0));
23 end entity;
24
25 architecture behavior of Imm is
26   -- sinais de decodificação do imediato, separado por extensão de sinal ou 0
27   signal mux_imm_sig : std_logic_vector(1 downto 0);
28   signal mux_imm_zero: std_logic_vector(1 downto 0);
```

```
29
30 begin
31   -- separação do sinal de decodificação por tipo de extensão
32   mux_imm_zero <= Imm_op(3 downto 2);
33   mux_imm_sig <= Imm_op(1 downto 0);
34   -- sinal imediato utilizado na instr. c.LUI
35   imm_lui(31 downto 18) <= (others => instr(12));
36   imm_lui(17 downto 12) <= instr(12)&instr(6 downto 2);
37   imm_lui(11 downto 0)  <= (others => '0');
38 ----------------------------------------
39 -- Immediate com extensão de sinal
40 ----------------------------------------
41   imm_sigext(31 downto 11) <= (others => instr(12)); -- MSb sempre no [12]
42
43   imm_sigext(10) <= instr(8) when (mux_imm_sig = "10") else  -- jump
44             instr(12); -- sinal extendido do MSb sempre no [12]
45
46   imm_sigext(9)  <= instr(10)when (mux_imm_sig = "10") else  -- jump
47             instr(12); -- sinal extendido do MSb sempre no [12]
48
49   imm_sigext(8)  <=  instr(9) when (mux_imm_sig = "10") else  -- jump
50             instr(4) when (mux_imm_sig = "01") else  -- addi16sp
51             instr(12); -- sinal extendido do MSb sempre no [12]
52
53   imm_sigext(7)  <=  instr(6) when (mux_imm_sig(1) = '1') else -- jump e branch
54             instr(3) when (mux_imm_sig(0) = '1') else -- addi16sp
55             instr(12); -- sinal extendido do MSb sempre no [12]
56
57   imm_sigext(6)  <=  instr(5) when (mux_imm_sig(0) = '1') else -- addi16sp e
       branch
58             instr(7) when (mux_imm_sig(1) = '1') else -- jump
59             instr(12); -- sinal extendido do MSb sempre no [12]
60
61   imm_sigext(5)  <=  instr(12) when (mux_imm_sig = "00") else
62             instr(2);         -- addi16sp, branch e jump
63
64   imm_sigext(4)  <= instr(6) when (mux_imm_sig(1) = '0') else
65             instr(11);          -- branch e jump
66
67   imm_sigext(3) <=  instr(10) when (mux_imm_sig = "11") else -- branch
68             '0'  when (mux_imm_sig = "01") else  -- addi16sp
```

```
69            instr(5);
70
71    imm_sigext(2 downto 1) <=  "00" when (mux_imm_sig = "01") else -- "00" quando
          addi16sp
72                  instr(4 downto 3);
73
74    imm_sigext(0) <=  instr(2) when (mux_imm_sig = "00") else '0'; -- '0' quando
          addi16sp, branch ou jump
75  ----------------------------------------
76  -- Immediate com extensão de 0s
77  ----------------------------------------
78    imm_zeroext(31 downto 10)  <= (others => '0');     -- extensao com 0s
79
80    imm_zeroext(9 downto 8) <=  instr(10 downto 9) when (mux_imm_zero = "11") else
            -- addi4spn
81                  "00";
82
83    imm_zeroext(7) <=  instr(8) when (mux_imm_zero(1) = '1') else -- swsp e
          addi4spn
84              instr(3) when (mux_imm_zero(0) = '1') else -- lwswp
85              '0';
86
87    imm_zeroext(6) <=  instr(7) when (mux_imm_zero(1) = '1') else -- swsp e
          addi4spn
88              instr(2) when (mux_imm_zero(0) = '1') else -- lwsp
89              instr(5);         -- lw e sw
90
91    imm_zeroext(5) <=  instr(12);
92
93    imm_zeroext(4) <=  instr(6) when (mux_imm_zero = "01") else -- lwsp
94              instr(11);         -- lw, sw, addi4spn e swsp
95
96
97    imm_zeroext(3) <=  instr(5) when (mux_imm_zero(0) = '1') else -- addi4spn,
          lwsp
98              instr(10);         -- lw, sw, swsp
99
100   imm_zeroext(2) <=  instr(4) when (mux_imm_zero = "01") else -- lwsp
101             instr(9) when (mux_imm_zero = "10") else -- swsp
102             instr(6);         -- addi4spn, lw e sw
103
```

```
104   imm_zeroext(1 downto 0) <= "00";
105
106 end architecture behavior;
```

E.5 – PROGRAMCOUNTER.VHD

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use work.core_pkg.all;
 4
 5 -- @file ProgramCounter.vhd
 6 -- @author Kevin Morais (moraiskv@gmail)
 7 -- @date 2017
 8 -- @version 1
 9 -- Contador de Programa, a cada instr. executada seu valor (PC) éincrementado
10 -- por +2 externamente, armazenando como a entrada PC_next. Possuí reset
11 -- síncrono, e sua hailitação de escrita écontrolada pelo sinal PC_we, ativo em
       1.
12 entity ProgramCounter is
13     port(clk : in std_logic;
14         rst : in std_logic;
15         PC_we : in std_logic;
16         PC_next : in std_logic_vector(n-1 downto 0);
17         PC : out std_logic_vector(n-1 downto 0));
18 end entity ProgramCounter;
19
20 architecture behavior of ProgramCounter is
21 begin
22
23   process (rst , clk , PC_we)
24   begin
25         if (rising_edge(clk)) then
26             if (rst = '1') then
27                 PC <= (others => '0'); -- valor de PC após o reset
28             elsif (PC_we = '1') then
29                 PC <= PC_next;
30             end if;
31     end if;
32   end process;
33
```

```
34 end architecture behavior;
```

E.6 – REGFILE.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.core_pkg.all;
5
6  -- @file RegFile.vhd
7  -- @author Kevin Morais (moraisku@gmail)
8  -- @date 2017
9  -- @version 1
10 -- Formado por 16 registradores de 4 bytes, x0 a x15, sendo x0 a constante 0
11 -- (ignorado por escritas), x1 o link register e x2 o stack pointer reg. Escreve
12 -- quando se tem RegFile_W = 1, de forma síncrona, e realiza leituras assí
       ncronras.
13 -- Os endereços de entrada (address_A/B/W) passa pela lógica do sinal RegAcess
14 -- que limita o acesso aos últimos 8 regs. quando necessário pela instrução.
15 -- O sinal RegDst_A impede o uso do sinal RegAcess para limitação de acesso
16 -- na execução da instr. c.ADDI4SPN. Pois esse mesmo sinal indica o uso
17 -- do end. 2 (SP reg), externamente ao banco, que sobre influéncia de RegAcess
18 -- iria gerar erro ao torna-lo 1010 = 10.
19 entity RegFile is
20    PORT(clk : in std_logic;
21         rst : in std_logic;
22         RegFile_W : in std_logic;
23         RegAcess : in std_logic;
24         RegDst_A : in std_logic;
25         address_A : in std_logic_vector(3 downto 0);
26         address_B : in std_logic_vector(3 downto 0);
27         address_W : in std_logic_vector(3 downto 0);
28         data_in_RF : in std_logic_vector(n-1 downto 0);
29         data_out_A : out std_logic_vector(n-1 downto 0);
30         data_out_B : out std_logic_vector(n-1 downto 0));
31 end entity RegFile;
32
33 architecture behavior of RegFile is
34
35    constant nE : integer := 16; --! num de regs. do banco, base E do RISC-V
```

```
36
37   type reg_array is ARRAY(0 to nE-1) of std_logic_vector(n-1 downto 0);
38   signal Register_File : reg_array;
39
40   -- endereços de acesso do banco de regs. após a lógica de RegAcess
41   signal addrA : std_logic_vector(3 downto 0);
42   signal addrB : std_logic_vector(3 downto 0);
43   signal addrW : std_logic_vector(3 downto 0);
44   signal acess_A : std_logic; -- sinal da lógica do sinal RegAcess
45
46 begin
47   -- limita o uso aos regs x8-x15 ou não
48   acess_A <= '1' when (RegAcess = '1' AND RegDst_A = '0') else '0';
49   addrA  <= (acess_A OR address_A(3))&address_A(2 downto 0);
50   addrB  <= (RegAcess OR address_B(3))&address_B(2 downto 0);
51   addrW  <= (RegAcess OR address_W(3))&address_W(2 downto 0);
52
53   -- leitura assíncrona do banco de regs.
54   data_out_A <= Register_File(to_integer(unsigned(addrA)));
55   data_out_B <= Register_File(to_integer(unsigned(addrB)));
56
57   -- escrita no banco, ignora o endereço 0, pois éa constante 0
58   process (clk, rst, RegFile_W , addrW)
59   begin
60     if (rising_edge(clk)) then
61             if (rst = '1') then
62         Register_File <= (others => (others => '0'));
63
64             elsif (RegFile_W = '1' AND addrW /= "0000") then
65         Register_File(to_integer(unsigned(addrW))) <= data_in_RF;
66
67             end if;
68         end if;
69   end process;
70
71 end architecture behavior;
```

E.7 – UN_CONTROL.VHD

```
1 library ieee;
```

```
2  use ieee.std_logic_1164.all;
3
4  -- @file Un_Control.vhd
5  -- @author Kevin Morais (moraisku@gmail)
6  -- @date 2017
7  -- @version 1
8  -- @brief Unidade de Controla do Processador
9  -- Unidade de controle, decodifica a instrução com base nos campos opcode
10 -- e funct3 da instr., [15:13|1:0]. Possui diversos sinais de saída
11 -- espalhados pelo núcleo. Instruções como c.LUI e c.ADDI16SP possuem
12 -- o mesmo funct3-opcode, logo se utiliza o sinal provido pela subunidade
13 -- de controle para determinar qual será executada, sinal referenciado
14 -- por addi16sp_en (ativo em '1' para ADDI16SP, do contrário, executa
15 -- c.LUI). O mesmo se aplica as instruções c.MV, c.ADD, c.JR e c.JALR,
16 -- diferenciados pelos sinais jr_en entre de desvio ou não, e então
17 -- na segunda diferenciação (entre c.MV e c.ADD, para jr_en = '0')
18 -- se utiliza o campo funct4 para determinar a instr. a executar.
19 -- O campo funct4 corresponde ao bit 12 da instr. O sinal ft2_temp
20 -- define o estado entre C e E, de operações da ALU entre Reg-Reg,
21 -- ou Reg-Imm, sinal provido também pela subunidade de controle.
22 entity Un_Control is
23   port(clk :  in std_logic;
24     rst :  in std_logic;
25     wait_mem : in std_logic;
26     addi16sp_en :  in std_logic;      --! sinal da subunidade de controle,
          utilizado para decodificar a instrução c.ADDI16SP
27     jr_en :  in std_logic;      --! sinal da subunidade de controle, utilizado
          para decodificar as instruções c.JR e c.JALR
28     ft2_temp :  in std_logic;      --! sinal da subunidade de controle,
          utilizado para diferenciar o estado C do E, em instruções MISC ALU
29     funct4 :  in std_logic;      --! bit 12 da instrução, utilizado para
          decodificar c.MV, c.ADD, c.JR e c.JALR
30     funct3_op :  in std_logic_vector(4 downto 0); --! campo da instrução a ser
          decodificado, funct3 e opcode concatenados
31     Imm_op :  out std_logic_vector(3 downto 0);
32     PC_W :  out std_logic;
33     PC_W_cond :  out std_logic;
34     PCFonte :  out std_logic;
35     RegFile_W :  out std_logic;
36     RegDst_A :  out std_logic;
37     RegDst_W :  out std_logic_vector(1 downto 0);
```

```
38      RegAcess :  out std_logic;
39      MemToReg :  out std_logic_vector(1 downto 0);
40      IouD :  out std_logic;
41      Instr_W :  out std_logic;
42      Mem_RW :  out std_logic;
43      Valid_addr :  out std_logic;
44      ALUop :  out std_logic_vector(1 downto 0);
45      ALUoprA :  out std_logic_vector(1 downto 0);
46      AlUoprB :  out std_logic_vector(1 downto 0));
47  end entity Un_Control;
48
49  architecture behavior of Un_Control is
50    -- estados da máquina de mealy
51    type estado_m is (O , A , B , C , D , E , F , G , H, I, J, K, L);
52
53    signal estado : estado_m; -- estado atual da maq. de estados
54    -- campos a serem decodificados da instr.
55    signal funct3  : std_logic_vector(2 downto 0);
56    signal opcode  : std_logic_vector(1 downto 0);
57    -- sinais de decodificação do campo embaralho do imediato
58    signal mux_imm_zero : std_logic_vector(1 downto 0); -- 2 MSB do sinal Imm_op
59    signal mux_imm_sig  : std_logic_vector(1 downto 0); -- 2 LSB do sinal Imm_op
60
61  begin
62    funct3 <= funct3_op(4 downto 2);
63    opcode <= funct3_op (1 downto 0);
64
65  -- Sinais de Controle de acordo com o estado e o sinal de entrada, Máquina de
        Mealy
66    -- controla o multiplexador de entrada de dados do PC
67    PCFonte <= '1' when (estado = B OR estado = J OR estado = K) else '0';
68    -- habilitação de escrita condicional do PC (usado em branchs)
69    PC_W_cond <= '1' when (estado = K) else '0';
70    -- habilitação de escrita do PC
71    PC_W <= '1' when (estado = B OR estado = J OR estado = L) else '0';
72    -- controla o multiplexador de entrada de endereço da memória (entre PC e
          RegALU)
73    IouD <= '1' when (estado = H OR estado = I) else '0';
74    -- habilitação de leitura/escrita da memória
75    Mem_RW <= '0' when (estado = H) else '1';
76    -- habilitação de escrita do regs. de instrução do núcleo
```

```
77  Instr_W <= '1' when (estado = A) else '0';
78  -- sinal que limita o acesso aos regs. x8-x15 do banco
79  RegAcess <= '1' when (opcode = "00" OR (funct3 > "011" AND opcode = "01"))
        else '0';
80  -- habilitação de escrita no banco de regs.
81  RegFile_W <= '1' when (estado = D OR (estado = J AND funct3 = "001") OR (
        estado = L AND funct4 = '1')) else '0';
82  -- sinal que controla o end. de entrada A do banco (entre addr_A/rs1 e a
        constante 2)
83  RegDst_A <= '1' when (funct3_op = "00000" OR ((funct3 = "010" OR funct3 = "110
        ") AND (opcode = "10"))) else
84        '0';
85  -- sinal que controla o MUX do end. de escrita (escolhe entre addressA,
        addressB ou a constante 1)
86  RegDst_W <= "01" when (opcode = "00") else
87        "10" when (estado = J OR estado = L) else
88        "00";
89  -- escolhe entre o Reg. de Dados, a saída da ALU e o immediate, como entrada
        do Reg File
90  MemToReg <= "00" when (estado = D AND (funct3 = "010" AND opcode(0) = '0'))
        else -- RegData
91        "10" when (estado = J OR estado = L)        else -- PC
92        "11" when (funct3_op = "01101" AND addi16sp_en = '0')    else -- c.LUI
93        "01";                -- RegALU
94  -- sinal de controle de operação da ALU, em ordem: MISC ALU, SLLI, BNEZ e ADD/
        BEQZ
95  ALUop <= "00" when ((estado = C AND opcode = "01") OR (estado = E AND funct3 =
         "100")) else
96        "01" when (estado = E AND opcode = "10")          else
97        "10" when (estado = K AND funct3 = "111")           else
98        "11";
99  -- escolhe o operando A da ALU, entre Reg. A, PC e 0
100 ALUoprA <= "00" when (estado = A OR estado = B OR estado = J) else -- PC
101       "10" when (estado = F) else        -- 0
102       "01";             -- Reg A
103 -- escolhe o operando B da ALU, em ordem: 2, imm_sigext, imm_zeroext, Reg B
104 ALUoprB <= "01" when (estado = A) else
105       "10" when (estado = B OR estado = E OR (estado = F AND opcode = "01") OR
              estado = J) else
106       "11" when (estado = G) else
107       "00";
```

```
108  -- decodificação do sinal imediato com MSB estendido
109  mux_imm_sig <=  "11" when (estado = B AND funct3(2 downto 1) = "11") else
110         "10" when (estado = B)        else
111         "01" when (estado = E AND funct3 = "011") else -- imm de c.ADDISP16
112         "00";
113  -- decodificação do sinal imediato completado com zeros
114  mux_imm_zero<= "01" when (estado = G AND funct3_op = "01010")  else -- c.LWSP
115         "10" when (estado = G AND funct3_op = "11010")  else -- c.SWSP
116         "11" when (estado = G AND funct3 = "000")  else -- c.ADDI4SPN
117         "00";              -- c.LW/SW
118  -- concatenação dos sinais, para envia-los a unidade Imm
119  Imm_op <= mux_imm_zero&mux_imm_sig;
120  -- indica quando um end. no barramento évalido, no estado '1'
121  Valid_addr <=  '1' when (estado = A OR estado = I OR estado = H) else
122         '0';
123
124  -- Máquina de estados de mealy
125  Maquina_de_Estados: process( clk , rst , wait_mem)
126  begin
127    if (rst = '1') then
128      estado <= 0; -- estado inicial do sistema
129
130    elsif (rising_edge(clk) AND wait_mem = '0') then
131      CASE estado is
132        -- estado inicial de busca de instrução e leitura do banco de reg.,
133        -- estados inicial comum a todas as instr.
134        when A =>  estado <= B;
135        when B =>
136          --- Quadrante 0, Q0
137          if (opcode = "00") then
138            estado <= G; -- lw , sw , addi4spn
139
140          --- Quadrante 1, Q1
141          elsif (opcode = "01") then
142
143            if (funct3 = "000") then -- addi , nop
144              estado <= E;
145
146            elsif (funct3 = "010") then -- li
147              estado <= F;
148
```

```
149           elsif (funct3 = "011" AND addi16sp_en = '0') then -- lui
150             estado <= D;
151
152           elsif (funct3 = "011" AND addi16sp_en = '1') then -- addi16sp
153             estado <= E;
154
155           elsif (funct3 = "100") then
156             if (ft2_temp = '1') then
157               estado <= C; -- sub, xor, or, and
158             else
159               estado <= E; -- srli, srai, andi
160             end if;
161
162           elsif (funct3 = "101" OR funct3 = "001") then -- j , jal
163             estado <= J;
164
165           elsif (funct3 = "110" OR funct3 = "111") then -- beqz , bneqz
166             estado <= K;
167           end if;
168
169         --- Quadrante 2, Q2
170         elsif (opcode = "10") then
171
172           if (funct3 = "000") then -- slli
173             estado <= E;
174
175           elsif (funct3 = "100" AND jr_en = '0') then
176             if (funct4 = '0') then
177               estado <= F; -- mv
178             else
179               estado <= C;  -- add
180             end if;
181
182           elsif (funct3 = "100" AND jr_en = '1') then -- jr , jalr
183             estado <= L;
184
185           elsif (funct3 = "010" OR funct3 = "110") then -- lwsp , swsp
186             estado <= G;
187           end if;
188
189         end if;
```

```
190
191         --- estados das instruções SUB, XOR, OR, AND e ADD:
192         -- operação Register to Register
193         when C => estado <= D;
194
195         --- estados das instruções ANDI, ADDI, ADDI16SP, SRLI, SRAI e SLLI:
196         -- operação Register to Immediate
197         when E => estado <= D;
198
199         --- estado da instrução MV e LI
200         when F => estado <= D;
201
202         --- estados das operações LW, SW, LWSP, SWSP e ADDI4SPN
203         when G => if (funct3 = "000") then
204                     estado <= D;  -- addi4spn
205                   elsif (funct3 = "110") then
206                     estado <= H;  -- sw, swsp
207                   elsif (funct3 = "010") then
208                     estado <= I;  -- lw, lwsp
209                   end if;
210
211         when H => estado <= A;
212
213         when I => estado <= D;
214
215         --- estado de desvio incondicional j e jal
216         when J => estado <= A;
217
218         --- estado de desvio condicional beqz e bnez
219         when K => estado <= A;
220
221         --- estado de desvio incondicional jr e jalr
222         when L => estado <= A;
223
224         --- estado de escrita no Banco de Registradores
225         when D => estado <= A;
226
227         --- estado de repouso do sistema
228         when O => estado <= A;
229
230       end CASE;
```

```
231     end if;
232   end process Maquina_de_Estados;
233 
234 end architecture behavior;
```

E.8 – TESTBENCH.VHD

```
  1 library ieee;
  2 use ieee.std_logic_1164.all;
  3 use ieee.std_logic_unsigned.all;
  4 use ieee.numeric_std.all;
  5 
  6 entity testbench is
  7   generic(n    : integer := 32; -- tamanho do dado
  8       n_addrIO : integer := 16); -- tamanho do end. de acesso dos disp. IO
  9 end entity testbench;
 10 
 11 architecture behavior of testbench is
 12 
 13   -- Núcleo Multicíclico RVC
 14   COMPONENT core
 15     port(clk  : in std_logic;
 16       rst   : in std_logic;
 17       wait_mem : in std_logic;
 18       we_l  : out std_logic;
 19       Valid_addr : out std_logic;
 20       addr_CPU : out std_logic_vector(n-1 downto 0);
 21       data_CPU : inout std_logic_vector(n-1 downto 0));
 22   end COMPONENT core;
 23 
 24   -- Componente Decodificador da Memória
 25   COMPONENT DEC is
 26         port (rst : in std_logic;
 27             clk : in std_logic;
 28             we_l : in std_logic;
 29             Valid_addr : in std_logic;
 30             csIO : out std_logic;
 31             wait_mem : out std_logic;
 32             addr_mem : in std_logic_vector(n-1 downto 0);
 33             data_MEM : inout std_logic_vector(n-1 downto 0));
```

```
34    end COMPONENT DEC;
35
36  -- Componente IO
37  COMPONENT IO is
38       port(rst : in std_logic;
39           clk : in std_logic;
40           csIO : in std_logic;
41           we_l : in std_logic;
42           addr : in std_logic_vector(n_addrIO-1 downto 0);
43           dataIO : inout std_logic_vector(n-1 downto 0);
44           RxD : in std_logic;
45           TxD : out std_logic);
46    end COMPONENT IO;
47
48 -- Sinais
49  signal clk   : std_logic;      -- sinal de clock do sistema
50  signal rst   : std_logic;      -- sinal de reset do sistema, ativo em '1'
51  signal wait_mem : std_logic;      -- sinal de entrada da CPU que indica quando
        o dado da memória esta pronto
52  signal Valid_addr : std_logic;      -- sinal que indica quando um endereço do
       barramento évalido
53  signal we_l  : std_logic;      -- sinal write enable pela CPU
54  signal addr_BUS : std_logic_vector(n-1 downto 0); -- barramento de endereço do
        sistema, largura de 32 bits
55  signal data_BUS : std_logic_vector(n-1 downto 0); -- barramento de dados do
       sistema, largura de 32 bits
56
57  signal csIO : std_logic;
58  signal RxD  : std_logic;
59  signal TxD  : std_logic;
60
61 begin
62
63  RxD <= '1';
64  TxD <= '1';
65
66  -- Processo de geração do sinal de clock de 100 MHz do tb.
67  signal_clk : process
68  begin
69
70    clk <= '0';  wait for 5 ns;
```

```
71    loop
72      clk <= '1'; wait for 5 ns;  -- Periodo = 10 ns , Freq = 100 MHz , Periodo/2
            = 5 ns
73      clk <= '0'; wait for 5 ns;
74    end loop;
75
76  end process signal_clk;
77
78  --! @brief Processo de geração do sinal de reset
79  signal_rst : process
80  begin
81
82    rst <= '0';   wait for 16 ns;
83    loop
84      rst <= '1';  wait for 10 ns;
85      rst <= '0';  wait for 10000000 ns;
86      rst <= '0';  wait for 10 ms;
87
88    end loop;
89  end process signal_rst;
90
91 -- Port Map
92   -- instanciação do processador
93   core_i: core port map
94   (rst  => rst ,
95   clk  => clk ,
96   we_l => we_l ,
97   wait_mem => wait_mem ,
98   Valid_addr => Valid_addr ,
99   addr_CPU => addr_BUS ,
100  data_CPU => data_BUS);
101
102  -- instanciação da memória
103  DEC_i: DEC port map
104  (rst => rst ,
105  clk  => clk ,
106  we_l => we_l ,
107  Valid_addr => Valid_addr ,
108  csIO  => csIO ,
109  wait_mem => wait_mem ,
110  addr_mem => addr_BUS ,
```

```
111   data_MEM => data_BUS);
112
113   -- instanciação do bloco IO
114     IO_i: IO port map
115     (rst  => rst ,
116     clk  => clk,
117     csIO  => csIO ,
118     addr  => addr_BUS(n_addrIO-1 downto 0) ,
119     dataIO  => data_BUS ,
120     we_l  => we_l ,
121   RxD  => RxD ,
122   TxD  => TxD);
123
124 end architecture behavior;
```

E.9 – TOP_MODULE.VHD

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.std_logic_unsigned.all;
 4 use ieee.numeric_std.all;
 5
 6 entity top_module is
 7   generic(n   : integer := 32; -- tamanho do dado
 8       n_addrIO : integer := 16); -- tamanho do end. de acesso dos disp. IO
 9   port(clk  : in std_logic;  -- sinal de clock
10     rst_b  : in std_logic;  -- sinal de reset, passa por lógica de debouncing
11     RxD  : in std_logic;  -- sinal de recepção da UART
12     TxD  : out std_logic); -- sinal de transmissão da UART
13 end entity top_module;
14
15 architecture behavior of top_module is
16
17   -- Núcleo Multicíclico RVC
18   COMPONENT core
19     port(clk  : in std_logic;
20       rst   : in std_logic;
21       wait_mem : in std_logic;
22       we_l  : out std_logic;
23       Valid_addr : out std_logic;
```

```
24      addr_CPU : out std_logic_vector(n-1 downto 0);
25      data_CPU : inout std_logic_vector(n-1 downto 0));
26  end COMPONENT core;
27
28  -- Componente Decodificador da Memória
29  COMPONENT DEC is
30       port (rst : in std_logic;
31           clk : in std_logic;
32           we_l : in std_logic;
33           Valid_addr : in std_logic;
34           csIO : out std_logic;
35           wait_mem : out std_logic;
36           addr_mem : in std_logic_vector(n-1 downto 0);
37           data_MEM : inout std_logic_vector(n-1 downto 0));
38  end COMPONENT DEC;
39
40  -- Componente IO
41    COMPONENT IO is
42       port(rst : in std_logic;
43           clk : in std_logic;
44           csIO : in std_logic;
45           we_l : in std_logic;
46           addr : in std_logic_vector(n_addrIO-1 downto 0);
47           dataIO : inout std_logic_vector(n-1 downto 0);
48           RxD : in std_logic;
49           TxD : out std_logic);
50    end COMPONENT IO;
51
52 -- Sinais
53  signal wait_mem : std_logic;      -- sinal de entrada da CPU que indica quando
        o dado da memória esta pronto
54  signal Valid_addr : std_logic;      -- sinal que indicai quando um endereço no
        barramento évalido
55  signal we_l  : std_logic;      -- sinal write enable pela CPU
56  signal addr_BUS : std_logic_vector(n-1 downto 0); -- barramento de endereço do
        sistema, largura de 32 bits
57  signal data_BUS : std_logic_vector(n-1 downto 0); -- barramento de dados do
       sistema, largura de 32 bits
58  signal csIO  : std_logic;      -- sinal de hablitação da seção IO
59
60  signal rst : std_logic;
```

```
61
62   type State_Type is (S0, S1);
63   signal State : State_Type := S0;
64
65   signal DPB, SPB : STD_LOGIC;
66   signal DReg : STD_LOGIC_VECTOR (7 downto 0);
67
68 begin
69
70   -- Reset Button Debouncing
71   process (clk , rst_b)
72       variable SDC : integer;
73       constant Delay : integer := 50000;
74     begin
75       if clk'Event and clk = '1' then
76         -- Double latch input signal
77         DPB <= SPB;
78         SPB <= rst_b;
79
80         case State is
81           when S0 =>
82             DReg <= DReg(6 downto 0) & DPB;
83
84             SDC := Delay;
85
86             State <= S1;
87           when S1 =>
88             SDC := SDC - 1;
89
90             if SDC = 0 then
91               State <= S0;
92             end if;
93           when others =>
94             State <= S0;
95         end case;
96
97         if DReg = X"FF" then
98           rst <= '1';
99         elsif DReg = X"00" then
100          rst <= '0';
101        end if;
```

```
102       end if;
103   end process;
104
105 -- Port Map
106   -- instanciação do processador
107   core_i: core port map
108   (rst  => rst ,
109   clk  => clk ,
110   we_l => we_l ,
111   wait_mem => wait_mem ,
112   Valid_addr => Valid_addr ,
113   addr_CPU => addr_BUS ,
114   data_CPU => data_BUS);
115
116   -- instanciação da memória
117   DEC_i: DEC port map
118   (rst => rst ,
119   clk  => clk ,
120   we_l => we_l ,
121   Valid_addr => Valid_addr ,
122   csIO  => csIO ,
123   wait_mem => wait_mem ,
124   addr_mem => addr_BUS ,
125   data_MEM => data_BUS);
126
127   -- instanciação do bloco IO
128     IO_i: IO port map
129     (rst  => rst ,
130     clk  => clk ,
131     csIO  => csIO ,
132     addr  => addr_BUS(n_addrIO-1 downto 0) ,
133     dataIO  => data_BUS ,
134     we_l  => we_l ,
135   RxD  => RxD ,
136   TxD  => TxD);
137
138 end architecture behavior;
```

```
##Clock signal
Net "clk" LOC=V10 |IOSTANDARD=LVCMOS33;
Net "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;

# input pad group
INST "rst" TNM = input_pad_group;
INST "wait_mem" TNM = input_pad_group;
INST "data_CPU<0>" TNM = input_pad_group;
INST "data_CPU<1>" TNM = input_pad_group;
INST "data_CPU<2>" TNM = input_pad_group;
INST "data_CPU<3>" TNM = input_pad_group;
INST "data_CPU<4>" TNM = input_pad_group;
INST "data_CPU<5>" TNM = input_pad_group;
INST "data_CPU<6>" TNM = input_pad_group;
INST "data_CPU<7>" TNM = input_pad_group;
INST "data_CPU<8>" TNM = input_pad_group;
INST "data_CPU<9>" TNM = input_pad_group;
INST "data_CPU<10>" TNM = input_pad_group;
INST "data_CPU<11>" TNM = input_pad_group;
INST "data_CPU<12>" TNM = input_pad_group;
INST "data_CPU<13>" TNM = input_pad_group;
INST "data_CPU<14>" TNM = input_pad_group;
INST "data_CPU<15>" TNM = input_pad_group;
INST "data_CPU<16>" TNM = input_pad_group;
INST "data_CPU<17>" TNM = input_pad_group;
INST "data_CPU<18>" TNM = input_pad_group;
INST "data_CPU<19>" TNM = input_pad_group;
INST "data_CPU<20>" TNM = input_pad_group;
INST "data_CPU<21>" TNM = input_pad_group;
INST "data_CPU<22>" TNM = input_pad_group;
INST "data_CPU<23>" TNM = input_pad_group;
INST "data_CPU<24>" TNM = input_pad_group;
INST "data_CPU<25>" TNM = input_pad_group;
INST "data_CPU<26>" TNM = input_pad_group;
INST "data_CPU<27>" TNM = input_pad_group;
INST "data_CPU<28>" TNM = input_pad_group;
INST "data_CPU<29>" TNM = input_pad_group;
INST "data_CPU<30>" TNM = input_pad_group;
INST "data_CPU<31>" TNM = input_pad_group;
TIMEGRP "input_pad_group" OFFSET = IN 10 ns VALID 10 ns BEFORE "clk" RISING;
```

```
42
43 # output pad group
44 INST "we_l" TNM = output_pad_group;
45 INST "Valid_addr" TNM = output_pad_group;
46 INST "data_CPU<0>" TNM = output_pad_group;
47 INST "data_CPU<1>" TNM = output_pad_group;
48 INST "data_CPU<2>" TNM = output_pad_group;
49 INST "data_CPU<3>" TNM = output_pad_group;
50 INST "data_CPU<4>" TNM = output_pad_group;
51 INST "data_CPU<5>" TNM = output_pad_group;
52 INST "data_CPU<6>" TNM = output_pad_group;
53 INST "data_CPU<7>" TNM = output_pad_group;
54 INST "data_CPU<8>" TNM = output_pad_group;
55 INST "data_CPU<9>" TNM = output_pad_group;
56 INST "data_CPU<10>" TNM = output_pad_group;
57 INST "data_CPU<11>" TNM = output_pad_group;
58 INST "data_CPU<12>" TNM = output_pad_group;
59 INST "data_CPU<13>" TNM = output_pad_group;
60 INST "data_CPU<14>" TNM = output_pad_group;
61 INST "data_CPU<15>" TNM = output_pad_group;
62 INST "data_CPU<16>" TNM = output_pad_group;
63 INST "data_CPU<17>" TNM = output_pad_group;
64 INST "data_CPU<18>" TNM = output_pad_group;
65 INST "data_CPU<19>" TNM = output_pad_group;
66 INST "data_CPU<20>" TNM = output_pad_group;
67 INST "data_CPU<21>" TNM = output_pad_group;
68 INST "data_CPU<22>" TNM = output_pad_group;
69 INST "data_CPU<23>" TNM = output_pad_group;
70 INST "data_CPU<24>" TNM = output_pad_group;
71 INST "data_CPU<25>" TNM = output_pad_group;
72 INST "data_CPU<26>" TNM = output_pad_group;
73 INST "data_CPU<27>" TNM = output_pad_group;
74 INST "data_CPU<28>" TNM = output_pad_group;
75 INST "data_CPU<29>" TNM = output_pad_group;
76 INST "data_CPU<30>" TNM = output_pad_group;
77 INST "data_CPU<31>" TNM = output_pad_group;
78 INST "addr_CPU<0>" TNM = output_pad_group;
79 INST "addr_CPU<1>" TNM = output_pad_group;
80 INST "addr_CPU<2>" TNM = output_pad_group;
81 INST "addr_CPU<3>" TNM = output_pad_group;
82 INST "addr_CPU<4>" TNM = output_pad_group;
```

```
INST "addr_CPU<5>" TNM = output_pad_group;
INST "addr_CPU<6>" TNM = output_pad_group;
INST "addr_CPU<7>" TNM = output_pad_group;
INST "addr_CPU<8>" TNM = output_pad_group;
INST "addr_CPU<9>" TNM = output_pad_group;
INST "addr_CPU<10>" TNM = output_pad_group;
INST "addr_CPU<11>" TNM = output_pad_group;
INST "addr_CPU<12>" TNM = output_pad_group;
INST "addr_CPU<13>" TNM = output_pad_group;
INST "addr_CPU<14>" TNM = output_pad_group;
INST "addr_CPU<15>" TNM = output_pad_group;
INST "addr_CPU<16>" TNM = output_pad_group;
INST "addr_CPU<17>" TNM = output_pad_group;
INST "addr_CPU<18>" TNM = output_pad_group;
INST "addr_CPU<19>" TNM = output_pad_group;
INST "addr_CPU<20>" TNM = output_pad_group;
INST "addr_CPU<21>" TNM = output_pad_group;
INST "addr_CPU<22>" TNM = output_pad_group;
INST "addr_CPU<23>" TNM = output_pad_group;
INST "addr_CPU<24>" TNM = output_pad_group;
INST "addr_CPU<25>" TNM = output_pad_group;
INST "addr_CPU<26>" TNM = output_pad_group;
INST "addr_CPU<27>" TNM = output_pad_group;
INST "addr_CPU<28>" TNM = output_pad_group;
INST "addr_CPU<29>" TNM = output_pad_group;
INST "addr_CPU<30>" TNM = output_pad_group;
INST "addr_CPU<31>" TNM = output_pad_group;
TIMEGRP "output_pad_group" OFFSET = OUT 10 ns AFTER "clk";
```

E.11 – RVC_SIST.UFC

```
##Clock signal
Net "clk" LOC=V10 |IOSTANDARD=LVCMOS33;
Net "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;

## Usb-RS232 interface
Net "RxD" LOC = N17 | IOSTANDARD=LVCMOS33; #Bank = 1, pin name =
    IO_L48P_HDC_M1DQ8, Sch name = MCU-RX
Net "TxD" LOC = N18 | IOSTANDARD=LVCMOS33; #Bank = 1, pin name = IO_L48N_M1DQ9,
```

```
    Sch name = MCU-TX
9
10 # Reset Switch
11 Net "rst_b" LOC = C9 | IOSTANDARD = LVCMOS33; #Bank = 0, pin name =
    IO_L34N_GCLK18, Sch name = BTND
12
13 # input pad group
14 INST "rst" TNM = input_pad_group;
15 INST "RxD" TNM = input_pad_group;
16 TIMEGRP "input_pad_group" OFFSET = IN 10 ns VALID 10 ns BEFORE "clk" RISING;
17
18 # output pad group
19 INST "TxD" TNM = output_pad_group;
20 TIMEGRP "output_pad_group" OFFSET = OUT 10 ns AFTER "clk";
```

APÊNDICE F – DESCRIÇÃO VHDL - NÚCLEO PIPELINE

Possuí dois arquivos de implementação, um para o núcleo apenas, e outro contando com todo o sistema. Dois arquivos VHDL de topo da hierarquia também, para testbench e implementação final.

F.1 – CORE.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.core_pkg.all;
5
6  -- @file core.vhd
7  -- @author Kevin Morais (moraiskv@gmail)
8  -- @date 2017
9  -- @version
10 -- Núcleo, arquivo VHDL de topo.
11 entity core is
12   port(clk : in std_logic;       -- sinal de clock
13     rst  : in std_logic;       -- sinal de reset, ativo em '1'
14     wait_data : in std_logic;       -- sinal da memória para dado pronto ('0')
         ou não ('1') quando realiza acesso
15     wait_instr : in std_logic;       -- sinal da ROM para dado (instr) pronto
         ('0') ou não ('1')
16     rvc  : in std_logic;       -- sinal que indica que a instr. buscada éda
         extensão C
17     sb_en : out std_logic;       -- store byte enable
18     sh_en : out std_logic;       -- store enable enable
19     we : out std_logic;       -- write enable, sinal de habilitação de escrita
         em componentes externos
20     re  : out std_logic;       -- read enable, sinal de habilitação de leitura
         de dados da memória, diferencia leitura de instr para dados na ROM
21     Valid_addr: out std_logic;       -- sinal de validade do endereço atual no
         barramento de dados, quando '1' = invalido
22     addr_CPU : out std_logic_vector(n-1 downto 0);  -- saída de endereço da CPU
23     data_CPU : inout std_logic_vector(n-1 downto 0)); -- saída/entrada de dados
         da CPU (barramento único e bidirecional)
```

```
24 end entity core;
25
26 architecture behavior of core is
27   --- Registradores de pipeline
28   signal IF_ID : std_logic_vector(n_IF_ID-1 downto 0);
29   signal ID_EX : std_logic_vector(n_ID_EX-1 downto 0);
30   signal EX_MEM: std_logic_vector(n_EX_MEM-1 downto 0);
31   signal MEM_WB: std_logic_vector(n_MEM_WB-1 downto 0);
32
33   -- EX
34   alias Mem_R_EX : std_logic     is ID_EX(148);
35   alias rs1_EX : std_logic_vector(3 downto 0) is ID_EX(11 downto 8);
36   alias rs2_EX  : std_logic_vector(3 downto 0) is ID_EX(7 downto 4);
37   alias rd_EX : std_logic_vector(3 downto 0) is ID_EX(3 downto 0);
38
39   -- MEM
40   alias RegFile_W_MEM : std_logic      is EX_MEM(147);
41   alias Branch_unc : std_logic     is EX_MEM(144);
42   alias Branch  : std_logic     is EX_MEM(143);
43   alias Mem_R_MEM  : std_logic      is EX_MEM(142);
44   alias Mem_W_MEM  : std_logic      is EX_MEM(141);
45   alias PC_MEM  : std_logic_vector(n-1 downto 0) is EX_MEM(139 downto 108);
46   alias addr_data : std_logic_vector(n-1 downto 0) is EX_MEM(75 downto 44);
47   alias rs2_MEM  : std_logic_vector(3 downto 0) is EX_MEM(7 downto 4);
48   alias rd_MEM  : std_logic_vector(3 downto 0) is EX_MEM(3 downto 0);
49
50   -- WB
51   alias RegFile_W_WB : std_logic      is MEM_WB(72);
52   alias rd_WB   : std_logic_vector(3 downto 0) is MEM_WB(3 downto 0);
53
54   --- Controle
55   signal PC_Src  : std_logic;
56   signal predict_flush: std_logic;
57   signal stall_IF_ID : std_logic;
58   signal stall_ID_IF : std_logic; -- sinal de stall provido no estagio ID para
         travar o estagio IF (sentido contrário do fluxo de dados, hazard)
59   signal stall_ID_IF_temp : std_logic;
60   signal flush_ID_EX : std_logic;
61   signal flush_EX_MEM : std_logic;
62
63   --- Forward
```

```
64    signal Forward_A : std_logic_vector(1 downto 0);
65    signal Forward_B : std_logic_vector(1 downto 0);
66    signal Forward_S : std_logic;
67    signal Forward_S_EX : std_logic;
68    signal fw_data_MEM : std_logic_vector(n-1 downto 0);
69    signal fw_data_WB : std_logic_vector(n-1 downto 0);
70
71    --- Dados
72    signal PC_IF   : std_logic_vector(n-1 downto 0);
73    signal PC_MEM_offset  : std_logic_vector(n-1 downto 0);
74    signal instr_IF   : std_logic_vector(n_I-1 downto 0);
75    signal Branch_addr_MEM : std_logic_vector(n-1 downto 0);
76    signal data_rd_MEM : std_logic_vector(n-1 downto 0);
77    signal data_rd_WB  : std_logic_vector(n-1 downto 0);
78
79    signal Mem_RW : std_logic;
80
81  begin
82
83  --------------------------
84  -- Barramentos de saída da CPU
85  --------------------------
86    instr_IF <= data_CPU(n-1 downto 0+length_dec); -- instrução buscada da memória
          no estágio IF
87
88    addr_CPU <= addr_data when (Mem_R_MEM = '1' OR Mem_W_MEM = '1') else -- endere
         ço enviado ao barramento
89          PC_IF;
90
91    we <= Mem_W_MEM;
92
93    re <= Mem_R_MEM;
94
95    Mem_RW <= Mem_R_MEM OR Mem_W_MEM;
96
97    Valid_addr <= predict_flush;
98
99  --------------------------
100 -- Lógica dos sinais de stall e flush
101 --------------------------
102   stall_IF_ID <= '0' when rst = '1' else
```

```
103         Mem_R_MEM OR Mem_W_MEM OR predict_flush;
104   stall_ID_IF <= '0' when (predict_flush = '1') else  -- stall_IF_ID tem
          prioridade sobre stall_ID_IF apenas na
105         stall_ID_IF_temp;     -- ocorrencia de stall por desvios (branch / jump
              )
106
107   flush_ID_EX <= '0' when rst = '1' else
108         predict_flush;
109
110   flush_EX_MEM <= '0' when rst = '1' else
111         predict_flush;
112
113 --------------------------
114 -- FORWARD
115 --------------------------
116     fw_data_MEM <= data_rd_MEM;
117     fw_data_WB <= data_rd_WB;
118
119 --------------------------
120 -- instancias
121 --------------------------
122   -- instanciação do estágio 1 do pipeline, Instruction Fetch
123   stage_1: stage1_IF
124   port map
125     (rst   => rst ,
126     clk   => clk ,
127     wait_data => wait_data ,
128     wait_instr  => wait_instr ,
129     Mem_RW    => Mem_RW ,
130     rvc   => rvc ,
131     stall_IF_ID => stall_IF_ID ,
132     stall_ID_IF => stall_ID_IF ,
133     Branch   => Branch ,
134     Branch_unc  => Branch_unc ,
135     PC_Src   => PC_Src ,
136     Branch_addr_MEM => Branch_addr_MEM ,
137     instr_IF   => instr_IF ,
138     predict_flush  => predict_flush ,
139     PC_IF  => PC_IF ,
140     PC_MEM   => PC_MEM ,
141     PC_MEM_offset  => PC_MEM_offset ,
```

```
142     IF_ID   => IF_ID
143     );
144
145   -- instanciao do estágio 2 do pipeline, Instruction Decode
146   stage_2: stage2_ID
147   port map
148     (rst  => rst ,
149     clk   => clk ,
150     wait_data => wait_data ,
151     Mem_R_EX  => Mem_R_EX ,
152     rd_EX   => rd_EX ,
153     rd_WB  => rd_WB ,
154     data_rd_WB  => data_rd_WB ,
155     IF_ID   => IF_ID ,
156     ID_EX  => ID_EX ,
157     stall_ID_IF => stall_ID_IF_temp ,
158     flush_ID_EX => flush_ID_EX ,
159     RegFile_W  => RegFile_W_WB
160     );
161
162   -- instanciação do estágio 3 do pipeline, Execute
163   stage_3: stage3_EX
164   port map
165     (rst   => rst ,
166     clk   => clk ,
167     wait_data => wait_data ,
168     flush_EX_MEM => flush_EX_MEM ,
169     ID_EX   => ID_EX ,
170     EX_MEM  => EX_MEM ,
171     Forward_A  => Forward_A ,
172     Forward_B  => Forward_B ,
173     Forward_S_EX => Forward_S_EX ,
174     fw_data_MEM => fw_data_MEM ,
175     fw_data_WB  => fw_data_WB
176     );
177
178   -- instanciação do estágio 4 do pipeline, Memory
179   stage_4: stage4_MEM
180   port map
181     (rst   => rst ,
182     clk   => clk ,
```

```
183     wait_data => wait_data ,
184     EX_MEM   => EX_MEM ,
185     PC_Src   => PC_Src ,
186     PC_MEM_offset => PC_MEM_offset ,
187     Branch_addr_MEM => Branch_addr_MEM ,
188     MEM_WB   => MEM_WB ,
189     Forward_S  => Forward_S ,
190     data_rd_MEm => data_rd_MEM ,
191     fw_data_WB  => fw_data_WB ,
192     sb_en   => sb_en ,
193     sh_en   => sh_en ,
194     data_BUS  => data_CPU
195     );
196
197   -- instanciação do estágio 5 do pipeline, Write Back
198   stage_5: stage5_WB
199   port map
200     (MEM_WB  => MEM_WB ,
201     data_rd_WB  => data_rd_WB
202     );
203
204   -- instanciação da unidade de forward, Forward Unit
205   forward: Forward_unit
206   port map
207     (RegFile_W_MEM  => RegFile_W_MEM ,
208     RegFile_W_WB  => RegFile_W_WB ,
209     rs1_EX  => rs1_EX ,
210     rs2_EX   => rs2_EX ,
211     rs2_MEM  => rs2_MEM ,
212     rd_MEM   => rd_MEM ,
213     rd_WB   => rd_WB ,
214     Forward_A  => Forward_A ,
215     Forward_B  => Forward_B ,
216     Forward_S  => Forward_S ,
217     Forward_S_EX => Forward_S_EX
218     );
219
220 end architecture behavior;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  package core_pkg is
5
6  ------------------------------------------------------------------
7  --- Constant
8  ------------------------------------------------------------------
9     constant length_dec : integer := 2;  -- num de bits utilizado na decodificaç
          ão de instr RVC, ignorando-os dentro do núcleo
10    constant b_offset : integer := 1;  -- tamanho do offset do PC a cada
          incremento, em bits, no caso 1, para instr tanto de 16 quatno 32 bits
11    --constant b_offset : integer := 2; -- para o uso da ISA base sem a ext C (
          RVC), apenas instr. de 32 bits
12    constant n : integer := 32; -- tamanho dos dados
13    constant n_E : integer := 16; -- num de regs. do banco, 16 de acordo com a
          base E do RISC-V
14    constant n_I : integer := n-length_dec; -- tamanho total da instr. ao
          ignorar os bits de decodificação RVC
15
16    constant n_IF_ID : integer := 63; -- tamanho do registrador de pipeline IF/
          ID
17    constant n_ID_EX : integer := 155; -- tamanho do registrador de pipeline ID/
          EX
18    constant n_EX_MEM : integer := 149; -- tamanho do registrador de pipeline EX
          /MEM
19    constant n_MEM_WB : integer := 73; -- tamanho do registrador de pipeline MEM
          /WB
20
21 ------------------------------------------------------------------
22 --- Componentes
23 ------------------------------------------------------------------
24  COMPONENT stage1_IF
25    port(rst   : in std_logic;
26      clk : in std_logic;
27           wait_data  : in std_logic;
28           wait_instr  : in std_logic;
29           Mem_RW : in std_logic;
30      rvc : in std_logic;
31      stall_IF_ID : in std_logic;
32      stall_ID_IF : in std_logic;
```

```
33      Branch : in std_logic;
34      Branch_unc : in std_logic;
35      PC_Src : in std_logic;
36      Branch_addr_MEM : in std_logiC_vector(n-1 downto 0);
37      instr_IF : in std_logic_vector(n_I-1 downto 0);
38      PC_MEM : in std_logic_vector(n-1 downto 0);
39      PC_MEM_offset  : in std_logic_vector(n-1 downto 0);
40      predict_flush : out std_logic;
41      PC_IF : out std_logic_vector(n-1 downto 0);
42      IF_ID : out std_logic_vector(n_IF_ID-1 downto 0));
43  end COMPONENT stage1_IF;
44
45  COMPONENT stage2_ID
46    port(rst  : in std_logic;
47      clk   : in std_logic;
48           wait_data : in std_logic;
49      RegFile_W : in std_logic;
50      Mem_R_EX : in std_logic;
51      rd_EX  : in std_logic_vector(3 downto 0);
52      rd_WB  : in std_logic_vector(3 downto 0);
53      data_rd_WB : in std_logic_vector(n-1 downto 0);
54      IF_ID  : in std_logic_vector(n_IF_ID-1 downto 0);
55      flush_ID_EX : in std_logic;
56      stall_ID_IF : out std_logic;
57      ID_EX  : out std_logic_vector(n_ID_EX-1 downto 0));
58  end COMPONENT stage2_ID;
59
60  COMPONENT stage3_EX
61    port(rst  : in std_logic;
62      clk   : in std_logic;
63           wait_data : in std_logic;
64      flush_EX_MEM: in std_logic;
65      ID_EX  : in std_logic_vector(n_ID_EX-1 downto 0);
66      Forward_A : in std_logic_vector(1 downto 0);
67      Forward_B : in std_logic_vector(1 downto 0);
68           Forward_S_EX: in std_logic;
69      fw_data_MEM : in std_logic_vector(n-1 downto 0);
70      fw_data_WB : in std_logic_vector(n-1 downto 0);
71      EX_MEM  : out std_logic_vector(n_EX_MEM-1 downto 0));
72  end COMPONENT stage3_EX;
73
```

```
74   COMPONENT stage4_MEM
75     port(rst   : in std_logic;
76       clk    : in std_logic;
77            wait_data : in std_logic;
78       EX_MEM   : in std_logic_vector(n_EX_MEM-1 downto 0);
79       Forward_S  : in std_logic;
80       fw_data_WB  : in std_logic_vector(n-1 downto 0);
81       PC_Src   : out std_logic;
82       PC_MEM_offset  : out std_logic_vector(n-1 downto 0);
83       Branch_addr_MEM : out std_logic_vector(n-1 downto 0);
84       MEM_WB   : out std_logic_vector(n_MEM_WB-1 downto 0);
85            sb_en : out std_logic;
86            sh_en : out std_logic;
87            data_rd_MEM  : out std_logic_vector(n-1 downto 0);
88       data_BUS   : inout std_logic_vector(n-1 downto 0));
89   end COMPONENT stage4_MEM;
90
91   COMPONENT stage5_WB
92     port(MEM_WB  : in std_logic_vector(n_MEM_WB-1 downto 0);
93       data_rd_WB : out std_logic_vector(n-1 downto 0));
94   end COMPONENT stage5_WB;
95
96  ------------------------------------------------------------------------
97   COMPONENT ProgramCounter
98     port( rst  : in std_logic;
99         clk  : in std_logic;
100        PC_we  : in std_logic;
101        PC_next : in std_logic_vector(n-1 downto 0);
102        PC   : out std_logic_vector(n-1 downto 0));
103  end COMPONENT ProgramCounter;
104
105  COMPONENT Predict_unit
106    port( clk    : in std_logic;
107        rst    : in std_logic;
108               wait_data : in std_logic;
109               wait_instr : in std_logic;
110               Mem_RW : in std_logic;
111        Branch   : in std_logic;
112        PC_Src   : in std_logic;
113        PC_next   : in std_logic_vector(n-1-b_offset downto 0);
114        PC_MEM   : in std_logic_vector(n-1-b_offset downto 0);
```

```
115       Branch_addr_MEM : in std_logic_vector(n-1 downto 0);
116       miss_predict : out std_logic;
117       taken   : out std_logic;
118       miss_check  : out std_logic;
119       addr_target  : out std_logic_vector(n-1 downto 0));
120   end COMPONENT Predict_unit;
121
122   COMPONENT RegFile
123       port(clk : in std_logic;
124           rst  : in std_logic;
125           RegFile_W : in std_logic;
126           rs1  : in std_logic_vector(3 downto 0);
127           rs2  : in std_logic_vector(3 downto 0);
128           rd  : in std_logic_vector(3 downto 0);
129           data_rd : in std_logic_vector(n-1 downto 0);
130           data_rs1 : out std_logic_vector(n-1 downto 0);
131           data_rs2 : out std_logic_vector(n-1 downto 0));
132   end COMPONENT RegFile;
133
134   COMPONENT Imm_unit
135     port(opcode : in std_logic_vector(4 downto 0);
136       imm_field : in std_logic_vector(n-1 downto 7);
137       imm_out : out std_logic_vector(n-1 downto 0));
138   end COMPONENT Imm_unit;
139
140   COMPONENT Control
141     port(opcode : in std_logic_vector(4 downto 0);
142       ctrl_ID : out std_logic_vector(10 downto 0));
143   end COMPONENT Control;
144
145   COMPONENT Hazards_unit
146     port(rs1_ID  : in std_logic_vector(3 downto 0);
147       rs2_ID  : in std_logic_vector(3 downto 0);
148       rd_EX  : in std_logic_vector(3 downto 0);
149       Mem_R_EX : in std_logic;
150       Mem_W_ID : in std_logic;
151       stall_ID_EX : out std_logic);
152   end COMPONENT Hazards_unit;
153
154   COMPONENT ALU
155     port(opr_A : in std_logic_vector(n-1 downto 0);
```

```
156       opr_B : in std_logic_vector(n-1 downto 0);
157       ALU_ctrl : in std_logic_vector(3 downto 0);
158       ALU_out : out std_logic_vector(n-1 downto 0);
159       ALU_zero : out std_logic);
160   end COMPONENT ALU;
161
162     COMPONENT Forward_unit
163     port(RegFile_W_MEM: in std_logic;
164       RegFile_W_WB : in std_logic;
165       rs1_EX  : in std_logic_vector(3 downto 0);
166       rs2_EX  : in std_logic_vector(3 downto 0);
167       rs2_MEM  : in std_logic_vector(3 downto 0);
168       rd_MEM  : in std_logic_vector(3 downto 0);
169       rd_WB  : in std_logic_vector(3 downto 0);
170       Forward_A : out std_logic_vector(1 downto 0);
171       Forward_B : out std_logic_vector(1 downto 0);
172       Forward_S : out std_logic;
173            Forward_S_EX : out std_logic);
174   end COMPONENT Forward_unit;
175
176 ------------------------------------------------------
177     COMPONENT BRAM_BHT is
178         generic( n_addr : integer ;
179                n_data : integer);
180         port( clk  : in  std_logic;
181                en   : in  std_logic;
182                we  : in  std_logic;
183                r_addr : in  std_logic_vector(n_addr-1 downto 0);
184                w_addr : in  std_logic_vector(n_addr-1 downto 0);
185                d_in : in  std_logic_vector(n_data-1 downto 0);
186                d_out : out std_logic_vector(n_data-1 downto 0));
187     end COMPONENT BRAM_BHT;
188
189   COMPONENT BRAM_target_BHT is
190     generic(n_addr : integer ;
191         n_data : integer);
192     port( clk  : in std_logic;
193                en : in std_logic;
194         we  : in std_logic;
195         r_addr : in std_logic_vector(n_addr-1 downto 0);
196         w_addr : in std_logic_vector(n_addr-1 downto 0);
```

```
197         d_in : in std_logic_vector(n_data-1 downto 0);
198         d_out : out std_logic_vector(n_data-1 downto 0));
199   end COMPONENT BRAM_target_BHT;
200
201 end package core_pkg;
```

F.3 – ALU.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5  use work.core_pkg.all;
6
7  -- @file ALU.vhd
8  -- @author Kevin Morais (moraisku@gmail)
9  -- @date 2018
10 -- @version 1
11 -- Aritmetic Logic Unit (ALU)
12 entity ALU is
13   port(opr_A : in std_logic_vector(n-1 downto 0); -- operando A
14     opr_B : in std_logic_vector(n-1 downto 0); -- operando B
15     ALU_ctrl : in std_logic_vector(3 downto 0);  -- sinal de controle que define
           a operação a ser realizada
16     ALU_out : out std_logic_vector(n-1 downto 0); -- saída resultante da operaçã
           o na ALU
17     ALU_zero : out std_logic); -- saída da ALU que indica result = 0
18 end entity ALU;
19
20 architecture behavior of ALU is
21
22     constant slt : std_logic_vector(n-1 downto 1) := (others => '0');
23
24   signal result : std_logic_vector(n-1 downto 0);
25
26   signal less_S : std_logic; -- '1' quando opr_A < opr_B, signed
27   signal less_U : std_logic; -- '1' quando opr_A < opr_B, unsigned
28
29   -- shift amount, utilizado nas instruções de shift, convertido para inteiro e
       depois bitvector,
```

```
30   -- a quantidade de bits a serem deslocados está definida no opr_B(4 downto 0);
31   signal shamt: integer range 0 to 31;
32
33 begin
34
35   -- converte para unsigned e depois para integer para se utilizar nos shifts (
         SRA, SRL e SLL)
36   shamt <= conv_integer(opr_B(4 downto 0));
37
38
39   -- comparador signed e unsigned, Set Less Than Signed/Unsigned
40     less_S <= '1' when (signed(opr_A) < signed(opr_B)) else
41               '0';
42     less_U <= '1' when (opr_A < opr_B) else
43               '0';
44
45   -- resultado
46   ALU_out <= result;
47   result <= opr_A + opr_B when ALU_ctrl = "0000" else
48                               opr_A - opr_B when ALU_ctrl = "0001" else
49 to_stdlogicvector(to_bitvector(opr_A) SLL shamt) when ALU_ctrl = "0010" else
50 to_stdlogicvector(to_bitvector(opr_A) SRL shamt) when ALU_ctrl = "0011" else
51                               opr_A XOR opr_B when ALU_ctrl = "0100" else
52 to_stdlogicvector(to_bitvector(opr_A) SRA shamt) when ALU_ctrl = "0101" else
53                               opr_A OR opr_B when ALU_ctrl = "0110" else
54                               opr_A AND opr_B when ALU_ctrl = "0111" else
55                               slt&less_S when ALU_ctrl = "1000" else
56                               slt&less_U when ALU_ctrl = "1001" else
57                               opr_B when ALU_ctrl = "1010" else
58                               opr_A + opr_B;
59
60   -- comparação a 0 do resultado
61   ALU_zero <= '1' when result = x"00000000" else
62         '0';
63
64 end architecture behavior;
```

F.4 – BRAM_BHT.VHD

```
1  library ieee;
```

```
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity BRAM_BHT is
6    generic( n_addr : integer ;
7        n_data : integer);
8    port( clk  : in  std_logic;
9        en   : in  std_logic;
10       we  : in  std_logic;
11       r_addr : in  std_logic_vector(n_addr-1 downto 0);
12       w_addr : in  std_logic_vector(n_addr-1 downto 0);
13       d_in : in  std_logic_vector(n_data-1 downto 0);
14       d_out : out std_logic_vector(n_data-1 downto 0));
15 end entity BRAM_BHT;
16
17 architecture behavior of BRAM_BHT is
18
19   constant ADDR_WIDTH : integer := n_addr;
20   constant DATA_WIDTH : integer := n_data;
21
22   type ram_type is ARRAY ((2**ADDR_WIDTH)-1 DOWNTO 0) of std_logic_vector(
         DATA_WIDTH-1 downto 0);
23   signal RAM : ram_type := (others => (others => '0'));
24
25 begin
26
27   process (clk , we)
28   begin
29     if (rising_edge(clk)) then
30       if (we = '1') then
31         RAM(conv_integer(w_addr)) <= d_in;
32
33       --else
34       elsif(en = '0') then
35         d_out <= RAM(conv_integer(r_addr));
36
37       end if;
38     end if;
39   end process;
40
41 end architecture behavior;
```

F.5 – BRAM_TARGET_BHT.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity BRAM_target_BHT is
6    generic( n_addr : integer ;
7        n_data : integer);
8    port( clk  : in std_logic;
9        en   : in std_logic;
10       we  : in std_logic;
11       r_addr : in std_logic_vector(n_addr-1 downto 0);
12       w_addr : in std_logic_vector(n_addr-1 downto 0);
13       d_in : in std_logic_vector(n_data-1 downto 0);
14       d_out : out std_logic_vector(n_data-1 downto 0));
15 end entity BRAM_target_BHT;
16
17 architecture behavior of BRAM_target_BHT is
18
19   constant ADDR_WIDTH : integer := n_addr;
20   constant DATA_WIDTH : integer := n_data;
21
22   type ram_type is ARRAY ((2**ADDR_WIDTH)-1 DOWNTO 0) of std_logic_vector(
         DATA_WIDTH-1 downto 0);
23   signal RAM : ram_type := (others => (others => '0'));
24
25 begin
26
27   process (clk , we)
28   begin
29     if (rising_edge(clk)) then
30       if (we = '1') then
31         RAM(conv_integer(w_addr)) <= d_in;
32
33       --else
34       elsif(en = '0') then
35         d_out <= RAM(conv_integer(r_addr));
36
37       end if;
38     end if;
```

```
39   end process;
40
41 end architecture behavior;
```

F.6 – CONTROL_UNIT.VHD

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3
 4 -- @file Control_unit.vhd
 5 -- @author Kevin Morais (moraiskv@gmail)
 6 -- @date 2018
 7 -- @version 1
 8
 9 -- Unidade de Controle: decodifica o campo opcode ( instr[6:2]), para definir o
       estado dos sinais
10 -- de controle no caminho de dados.
11 entity Control is
12   port(opcode : in std_logic_vector(4 downto 0);  -- campo opcode a ser
         decodificado
13      ctrl_ID : out std_logic_vector(10 downto 0)); -- sinais de controle
           restantes no estágio ID
14 end entity Control;
15
16 architecture behavior of Control is
17
18   -- valor do opcode de classes de instruçóes: branch, load, store;
19   -- ou então de instruçóes especificas: LUI, AUIPC, JALR.
20   constant lui_op : std_logic_vector(4 downto 0) := "01101";
21   constant auipc_op : std_logic_vector(4 downto 0) := "00101";
22   constant jal_op : std_logic_vector(4 downto 0) := "11011";
23   constant jalr_op : std_logic_vector(4 downto 0) := "11001";
24   constant branch_op : std_logic_vector(4 downto 0) := "11000";
25   constant load_op : std_logic_vector(4 downto 0) := "00000";
26   constant store_op : std_logic_vector(4 downto 0) := "01000";
27
28     -- sinais do estágio WB
29     alias RegFile_W : std_logic is ctrl_ID(10);
30     -- sinais do estágio MEM
31     alias RegFile_Src_0 : std_logic_vector(1 downto 0) is ctrl_ID(9 downto 8);
```

```
32      alias Branch_unc : std_logic is ctrl_ID(7);
33      alias Branch : std_logic is ctrl_ID(6);
34      alias Mem_R  : std_logic is ctrl_ID(5);
35      alias Mem_W : std_logic is ctrl_ID(4);
36      alias Branch_Src : std_logic is ctrl_ID(3);
37      -- sinais do estágio EX
38      alias ALU_op : std_logic_vector(1 downto 0) is ctrl_ID(2 downto 1);
39      alias ALU_oprB : std_logic is ctrl_ID(0);
40
41  begin
42      ------------------------------
43      -- Sinais do Estágio WB
44      ------------------------------
45      --'0' when (opcode = store_op OR opcode = branch_op) else
46      RegFile_W <= '0' when (opcode(3 downto 0) = "1000") else
47                   '1';
48
49      ------------------------------
50      -- Sinais do Estágio MEM
51      ------------------------------
52      RegFile_Src_0 <= "00" when (opcode = jal_op OR opcode = jalr_op) else
53                       "01" when (opcode = auipc_op) else
54                   "11";
55
56      --'1' when (opcode = jal_op OR opcode = jalr_op) else
57      Branch_unc <= '1' when (opcode = jal_op OR opcode = jalr_op) else
58                    '0';
59
60      --'1' when (opcode = branch_op) else
61      Branch <= '1' when (opcode(4 downto 3) = "11" AND opcode(0) = '0') else
62                '0';
63
64      -- o campo funct3 = [14:12], decidem entre LW/LH/LB, no estágio MEM e WB
65      Mem_R <= '1' when (opcode = load_op) else
66               '0';
67
68      -- o campo funct3 = [14:12], decidem entre SW/SH/SB, no estágio MEM
69      Mem_W <= '1' when (opcode = store_op) else
70               '0';
71
72      --'1' when (opcode = jalr_op) else
```

```
73     Branch_Src <= '1' when (opcode(1) = '0' AND opcode(0) = '1') else
74                   '0';
75
76     ------------------------------
77     -- Sinais do Estágio EX
78     ------------------------------
79     ALU_op <= "01" when (opcode = "00100" OR opcode = "01100") else -- ALU-
           funct3
80               "10" when (opcode = branch_op) else -- Branch-funct3
81               "11" when (opcode = lui_op) else -- LUI
82               "00"; -- ADD
83
84     ALU_oprB <= '0' when (opcode = "01100" OR opcode = branch_op) else
85                 '1';
86
87 end architecture behavior;
```

F.7 – FORWARD_UNIT.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  -- @file Forward_unit.vhd
5  -- @author Kevin Morais (moraisku@gmail)
6  -- @date 2018
7  -- @version 1
8  -- Unidade de Forward, detecta hazards de dados (depedencias verdadeiras, RAW),
       e implementa a tecnica de forwarding a fim de
9  -- evitar perdas de desempenho. A verificação se faz por meio da comparação do
       end. de registrador destino (rd) de um estagio
10 -- X, com um dos registradores dos operandos A e/ou B (rs1 e rs2) de um estagio
       X-1. Os estagios 'vigiados' pela unidade de
11 -- forward corresponde ao 3, 4 e 5 (EX, MEM e WB), o estagio 2 (ID) realiza o
       forward internamente no Banco de Registradores.
12 -- O forward também entra em ação no uso de instruçóes SW que necessitam de um
       dado ainda não escrito no banco de regs., mas
13 -- que já está disponível em um estágio seguinte.
14 entity Forward_unit is
15   port( RegFile_W_MEM: in std_logic;      -- habilitação de escrita no Banco de
         Regs., no estagio MEM
```

```
16      RegFile_W_WB : in std_logic;      -- habilitação de escrita no Banco de
            regs., no estagio WB
17      rs1_EX  : in std_logic_vector(3 downto 0);  -- endereço do regs. rs1 no
            estagio Execute
18      rs2_EX  : in std_logic_vector(3 downto 0);  -- endereço do regs. rs2 no
            estagio Execute
19      rs2_MEM  : in std_logic_vector(3 downto 0);  -- endereço do regs. rs2 no
            estagio Memory
20      rd_MEM  : in std_logic_vector(3 downto 0);  -- endereço do regs. rd no está
            gio Memory
21      rd_WB  : in std_logic_vector(3 downto 0);  -- endereço do regs. rd no está
            gio Write Back
22      Forward_A : out std_logic_vector(1 downto 0); -- sinal do MUX A do data
            path, no estágio EX
23      Forward_B : out std_logic_vector(1 downto 0); -- sinal do MUX B do data
            path, no estágio EX
24      Forward_S : out std_logic;      -- sinal do MUX S do data path, no estágio
            MEM
25      Forward_S_EX : out std_logic);      -- sinal do MUX S EX do data path, no
            estágio EX
26  end Forward_unit;
27
28  architecture Behavioral of Forward_unit is
29
30  begin
31    Forward_A <= "10" when  (RegFile_W_MEM = '1' AND rs1_EX = rd_MEM) else
32           "01" when (RegFile_W_WB = '1' AND rs1_EX = rd_WB) else
33           "00";
34
35    Forward_B <=  "10" when  (RegFile_W_MEM = '1' AND rs2_EX = rd_MEM) else
36           "01" when (RegFile_W_WB = '1' AND rs2_EX = rd_WB) else
37           "00";
38
39    Forward_S <=  '1' when (RegFile_W_WB = '1' AND rs2_MEM = rd_WB) else
40           '0'; -- when (RegFile_W_WB = '1' AND rs2_MEM \= rd_WB);
41
42    Forward_S_EX <= '1' when (RegFile_W_WB = '1' AND rs2_EX = rd_WB) else
43           '0'; -- when (RegFile_W_WB = '1' AND rs2_MEM \= rd_WB);
44
45  end Behavioral;
```

F.8 – HAZARDS_UNIT.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  -- Unidade de detecção de hazards.
5  -- Utilizada para detectar possíveis hazards de dados (dependencias verdadeiras)
       entre
6  -- instruções de LW seguidas por instruções aritméticas/lógicas que fazem uso do
       resultado
7  -- da execução da instrução LW. Verifica Mem_W_ID, pois em intr. de LW seguida
      por SW
8  -- não ocorre hazards, devido a técnica de fowarding implementada no estágio MEM
      .
9  entity Hazards_unit is
10   port( rs1_ID  : in std_logic_vector(3 downto 0); -- endereço rs1 do banco de
         regs. no estágio ID
11       rs2_ID  : in std_logic_vector(3 downto 0); -- endereço rs2 do banco de regs
            . no estágio ID
12       rd_EX  : in std_logic_vector(3 downto 0); -- endereço rd do banco de regs.
            no estágio EX
13       Mem_R_EX  : in std_logic;      -- sinal de controle Mem_R no estágio EX
14       Mem_W_ID : in std_logic;      -- sinal de controle Mem_W no estágio ID
15       stall_ID_EX : out std_logic);     -- sinal de stall do regs. de pipeline ID
            /EX
16 end entity Hazards_unit;
17
18 architecture behavior of Hazards_unit is
19 begin
20     stall_ID_EX <= '1' when (Mem_R_EX = '1' AND Mem_W_ID = '0' AND (rd_EX =
          rs1_ID OR rd_EX = rs2_ID)) else
21                    '0';
22
23 end architecture behavior;
```

F.9 – IMM_UNIT.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.core_pkg.all;
```

```
4
5  -- @file Imm.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2017
8  -- @version 1
9  -- Immediate da instrução
10 -- As instr. do RISC-V possuem seu campo imediato embaralhado ao longo de sua
       codificação, reduzindo o num de
11 -- multiplexadores para que se selecione os campos corretos. A extensão ésempre
       pelo sinal(Msb), que fica posicionado
12 -- de forma fixa no bit [31] da instrução. A base I possui cinco tipos
       diferentes de imediatos:
13 -- -> I - type, para operaçóes 'Registrador <-> Imediato', possui três opcode,
       "11001", "00100" e "00000";
14 -- -> S - type, para instr. de Store Word, possui um opcode, "01000";
15 -- -> B - type, para instr. de Branch Condicional, possui um opcode, "11000";
16 -- -> U - type, para as instruçóes LUI e AUIPC, possui dois opcode, "01101" e
       "00101";
17 -- -> J - type, para a instr. JAL, possui um opcode, "11011";
18 entity Imm_unit is
19   port(opcode  : in std_logic_vector(4 downto 0);  -- campo opcode da instrução
         para decodificação do campo imediato
20     imm_field : in std_logic_vector(n-1 downto 7); -- campos da instrução que
         possuem imediatos
21     imm_out  : out std_logic_vector(n-1 downto 0)); -- sinal imediato
         decodificado
22 end entity Imm_unit;
23
24 architecture behavior of Imm_unit is
25
26 begin
27 ----------------------------------------
28 -- Decodificação
29 ----------------------------------------
30     imm_out(31 downto 20)<= imm_field(31 downto 20) when (opcode = "01101" OR
         opcode = "00101") else-- U_type
31                             (others => imm_field(31));            -- J_type ;
                                I_type ; B_type ; S_type
32
33     imm_out(19 downto 12)<= imm_field(19 downto 12) when (opcode = "01101" OR
         opcode = "00101" OR opcode = "11011") else-- U_type ; J_type
```

```
34                          (others => imm_field(31));  -- I_type ; B_type ; S_type
35
36      imm_out(11) <= '0' when (opcode = "01101" OR opcode = "00101") else --
            U_type
37                          imm_field(20) when (opcode = "11011") else -- J_type
38                          imm_field(7) when (opcode = "11000") else -- B_type
39                          imm_field(31); -- I_type ; S_type
40
41      imm_out(10 downto 5)<= (others => '0') when (opcode = "01101" OR opcode = "
            00101") else -- U_type
42                          imm_field(30 downto 25); --I_type ; B_type ; S_type ;
                                J_type
43
44      imm_out(4 downto 1) <= (others => '0') when (opcode = "01101" OR opcode = "
            00101") else  -- U_type
45                          imm_field(11 downto 8) when (opcode = "11000" OR opcode
                                 = "01000") else -- B_type ; S_type
46                          imm_field(24 downto 21);  -- I_type ; J_type
47
48      imm_out(0) <= imm_field(20) when (opcode = "11001" OR opcode = "00100" OR
            opcode = "00000") else -- I_type
49                          imm_field(7) when (opcode = "01000") else -- S_type
50                          '0'; -- U_type , B_type , J_type
51
52  end architecture behavior;
```

F.10 – PREDICT_UNIT.VHD

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   use ieee.numeric_std.all;
5   use work.core_pkg.all;
6
7   -- @file Predict_unit.vhd
8   -- @author Kevin Morais (moraiskv@gmail)
9   -- @date 2018
10  -- @version 1
11  -- Preditor Dinãmico de Desvios
12  -- Consiste em uma tabela de ' 2**n_BHT ' linhas, sendo armazenada em cada uma
```

```
    informações
13 -- referentes a instruções branch: \n
14 -- -> 'tag', MSbits para identificar se a linha acessada na BHT possui informaçó
       es correspondentes a instrução atual;
15 -- -> 'target', endereço alvo do preditor, obtido após a primeira execução da
       instrução no pipeline;
16 -- -> 'state', dois bits de previsão que indicam a tentativa seguinte do
       preditor, funcionam com uma máquina de estados de mealy;
17 -- -> 'valid', bit de validade, quando '0' (valor inicial do sistema), indica
       que a linha não possu informações válidas;
18 entity Predict_unit is
19   port(rst  : in std_logic; -- sinal de reset do sistema
20     clk   : in std_logic; -- sinal de clk do sistema
21     wait_data  : in std_logic; -- sinal de espera por dado da memória (stall)
22     wait_instr : in std_logic; -- sinal de espera por instr. da memória (stall)
23     Mem_RW   : in std_logic;
24     -- entrada de sinais de controle
25     Branch  : in std_logic; -- sinal que indica que o desvio écondicional (
           poderia ser um jump)
26     PC_Src  : in std_logic; -- sinal que indica que o branch executado foi
           definido como taken (para '1')
27     -- saída de sinais de controle
28     miss_predict: out std_logic; -- sinal indicando que o preditor errou na ação
            (flush)
29     taken  : out std_logic; -- sinal indicando que a previsão da linha acessada
           éde taken (para = '1')
30     miss_check : out std_logic; -- sinal para verificar se o erro (miss) foi
           sobre a previsão 'taken' ou 'not taken'
31     -- entrada de endereço de PC_IF, PC_MEM, alvo do Branch e alvo do preditor
32     PC_next   : in std_logic_vector(n-1-b_offset downto 0); -- PC no estágio IF,
            para acessar a BHT
33     PC_MEM   : in std_logic_vector(n-1-b_offset downto 0); -- PC no estágio MEM
           para escrita na BHT
34     Branch_addr_MEM : in std_logic_vector(n-1 downto 0); -- end. alvo de branch
           no estágio MEM para escrita na BHT
35     addr_target  : out std_logic_vector(n-1 downto 0)); -- endereço alvo do
           preditor para o PC_next encontrado na BHT
36 end entity Predict_unit;
37
38 architecture behavior of Predict_unit is
39
```

```
40    constant n_BHT : integer := 9; -- tamanho da BHT (branch history table) em n
          úmero de linhas (entradas), 2**n_BHT
41    constant n_tag : integer := n-n_BHT-b_offset;-- tamanho da tag armazenada na
           BHT
42    constant n_PC : integer := n-b_offset; -- tamanho de PC (next ou MEM) ao
          descartar o byte offset
43
44  -- estados da máq. de mealy, 'S' e 'W' = 'Strong' e 'Weak', 'NT' e 'T' = 'Not
      Taken' e 'Taken'
45  type state_type IS (S_NT, W_NT, W_T, S_T);
46
47  signal bimodal_state : state_type;
48  signal we  : std_logic;      -- sinal de habilitação de escrita na tag_BRAM e
      na target_BRAM
49  signal index : std_logic_vector(n_BHT-1 downto 0); -- sinal que contém os lsb
      de PC_next, que são utilizados como index da BHT
50  signal tag_out : std_logic_vector(n_tag-1 downto 0); -- sinal de leitura da
      tag_BHT (BRAM_tag_BHT)
51  signal state_in : std_logic_vector(1 downto 0);  -- sinal de entrada da
      state_BHT (BRAM_state_BHT)
52  signal state_out: std_logic_vector(1 downto 0);  -- sinal de leitura da
      state_BHT (BRAM_state_BHT)
53  signal valid_in : std_logic; -- sinal de entrada da BHT dos bits de validade,
      fixo em '1'
54  signal valid_out : std_logic; -- sinal de leitura da valid_BHT (BRAM_valid_BHT
      )
55
56  signal miss  : std_logic; -- sinal que indica se ocorreu miss ou não na
      procura da BHT
57  signal branch_taken: std_logic; -- sinal que indica que o branch executado foi
           taken quando '1'
58  signal msb_state : std_logic; -- msb de previsão da linha acessada na BHT,
      propagado até o momento em que a instr. branch termina de ser executada
59  signal miss_BHT : std_logic; -- sinal de habilitação de escrita na BHT quando
      ocorre miss de procura da instr na BHT
60
61  -- registradores temporrios para armazenar o estado do branch (bits de estado,
           miss ou hit) do 1 ao 4 estágio, para
62  -- que na ocorréncia de miss, se possa corrigir os bits de previsão
63  signal temp1 : std_logic_vector(2 downto 0);
64  signal temp2 : std_logic_vector(2 downto 0);
```

```
65      signal temp3 : std_logic_vector(1 downto 0);
66
67    signal en  : std_logic;
68    signal count : std_logic;
69
70    -- BRAM2
71    signal bram_in : std_logic_vector(n_tag+3-1 downto 0);
72    signal bram_out : std_logic_vector(n_tag+3-1 downto 0);
73
74  begin
75    -- BRAM 2
76    bram_in(n_tag+2 downto n_tag+1) <= state_in;
77    bram_in(n_tag)     <= valid_in;
78    bram_in(n_tag-1 downto 0)   <= PC_MEM(n_PC-1 downto n_BHT);
79
80    state_out  <= bram_out(n_tag+2 downto n_tag+1);
81    valid_out  <= bram_out(n_tag);
82    tag_out  <= bram_out(n_tag-1 downto 0);
83
84    ----------------------------------------
85    -- lógica de habilitação (stall) pela laténcia de acesso de 1 ciclo da memória
86    en <= wait_data OR wait_instr OR Mem_RW;
87
88    valid_in <= '1';
89
90      -- atribui os lsb de PC_next para 'index', definindo o end. de acesso da BHT
91    index <= PC_next(n_BHT-1 downto 0);
92
93    -- sinal de habilitação da escrita na tag_BHT, target_BHT e valid_BHT, é
          habilitado quando:
94    --não se encontrou a instr. de branch na BHT, ou o bit de validade é= '0',
          desde que
95    -- Branch = '1'para indicar que éuma instr. de branch
96    we <= '1' when (Branch = '1' AND miss_BHT = '1') else
97        '0';
98
99    -- miss = '1' caso a linha acessada da BHT possua uma tag diferente de PC_next
          , ou
100   -- se o bit de validade for = '0', indicando miss na BHT então
101   miss <= '0' when ( ( tag_out = PC_next(n_PC-1 downto n_BHT) ) AND valid_out =
          '1' ) else
```

```
102     '1';
103
104   -- define se a previsão étaken ou not taken de acordo com o msb de 'state_out
        ', o
105   -- chute inicial de toda instr. de branch énot taken no estado SNT (strongly
        not taken)
106   taken <= state_out(1) AND (NOT(miss));
107
108   -----------------
109   -- faz a leitura dos dados de predição propagados até o momento em que o
        branch foi
110   -- executado no 4 estágio
111   -----------------
112   msb_state  <= temp3(0);
113   miss_BHT  <= temp3(1);
114
115   -----------------
116   -- Verifica se aconteceu hit ou miss no 4 estágio, e determina se o erro o
        ocorreu
117   -- sobre uma previsão taken ou not taken
118   -----------------
119   branch_taken <= PC_Src AND Branch; -- sinais de controle que indicam uma instr
         branch taken
120
121   miss_check <= '0' when (msb_state = '0' AND branch_taken = '1') else
122           '1' when (msb_state = '1' AND branch_taken = '0') else '0';
123
124   miss_predict <= '1' when (branch_taken /= msb_state) else -- XOR
125          '0';
126
127   -----------------
128   -- Propaga os valores dos bits de previsão (bit_state) da linha selecionada da
         BHT
129   -- e se ela foi encontrada ou não (miss_BHT), até o momento em que a instr. de
130   -- branch estiver no 4 estágio.
131   process(clk , rst , wait_data , en)
132   begin
133     if (rising_edge(clk)) then
134       if (rst = '1') then
135         temp1 <= (others => '0');
136         count <= '0';
```

```
137
138       elsif (en = '1' AND count = '0') then
139         --não escreve em temp1
140
141       elsif (count = '1') then
142         temp1 <= "100";
143         count <= NOT(count);
144
145       else
146         temp1(1 downto 0) <= state_out;
147         temp1(2) <= miss;
148         count <= NOT(count);
149
150       end if;
151     end if;
152   end process;
153
154   process(clk , rst , wait_data)
155   begin
156     if (rising_edge(clk)) then
157       if (rst = '1') then
158         temp2 <= (others => '0');
159         temp3 <= (others => '0');
160
161       elsif (wait_data = '1') then
162         -- não escreve
163
164       else
165         temp2(2 downto 0) <= temp1(2 downto 0);
166         temp3(1 downto 0) <= temp2(2 downto 1);
167
168       end if;
169     end if;
170   end process;
171
172   -----------------
173   -- Processo da máquina de estados do preditor
174   bimodal: process(clk , rst , branch_taken , temp2)
175   begin
176     if (rising_edge(clk)) then
177       if (rst = '1') then
```

```
178       bimodal_state <= S_NT;
179
180     elsif(temp2(1 downto 0) = "00") then
181       bimodal_state <= S_NT;
182
183     elsif(temp2(1 downto 0) = "01") then
184       bimodal_state <= W_NT;
185
186     elsif(temp2(1 downto 0) = "10") then
187       bimodal_state <= W_T;
188
189     elsif(temp2(1 downto 0) = "11") then
190       bimodal_state <= S_T;
191
192     end if;
193   end if;
194 end process bimodal;
195
196 -- saída da máquina de estados
197 state_in <= "00" when (((bimodal_state = S_NT) OR (bimodal_state = W_NT) OR (
        bimodal_state = W_T)) AND branch_taken = '0')else
198       "01" when (bimodal_state = S_NT AND branch_taken = '1')           else
199       "10" when (bimodal_state = S_T AND branch_taken = '0')           else
200       "11" when (((bimodal_state = W_NT) OR (bimodal_state = W_T) OR (
              bimodal_state = S_T)) AND branch_taken = '1')else
201       "00";
202
203 -------------------------------------------------------------
204 -- Instanciação da tabela como BRAM
205 -------------------------------------------------------------
206 -- instanciação de 'BRAM_BHT'
207 BRAM_BHT_i: BRAM_BHT
208 generic map
209 (n_addr => n_BHT ,
210 n_data => n_tag+3) -- tag + 1 bit validade + 2 bit estado
211 port map
212 (clk  => clk ,
213 en  => en ,
214 we   => Branch ,
215 w_addr  => PC_MEM(n_BHT-1 downto 0) , -- write address
216   r_addr => index ,
```

```
217   d_in => bram_in ,
218   d_out => bram_out
219   );
220
221   -- instanciação de 'target_BHT'
222   BRAM_2 : BRAM_target_BHT
223   generic map
224   (n_addr => n_BHT ,
225   n_data => n)
226   port map
227   (clk => clk ,
228   en => en ,
229   we => we ,
230     w_addr => PC_MEM(n_BHT-1 downto 0) , -- write address
231     r_addr => index ,
232   d_in => Branch_addr_MEM ,
233   d_out => addr_target
234   );
235
236 end architecture behavior;
```

F.11 – PROGRAMCOUNTER.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.core_pkg.all;
4
5  -- @file ProgramCounter.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2018
8  -- @version 1
9  -- Contador de Programa do processador, a cada ciclo atualiza seu valor por PC+
       incremento
10 -- ou algum desvio gerado, buscando a próxima instrução a ser executada.
11 entity ProgramCounter is
12     port( rst  : in std_logic; -- sinal de reset
13       clk  : in std_logic;       -- sinal de clock
14       PC_we  : in std_logic;       -- sinal de habilitação de escrita
15       PC_next : in std_logic_vector(n-1 downto 0); -- próximo valor de PC
16       PC   : out std_logic_vector(n-1 downto 0)); -- valor atual de PC
```

```
17 end entity ProgramCounter;
18
19 architecture behavior of ProgramCounter is
20 begin
21   -- Possui reset síncrono (rst = '1'), do contrário o seu valor éescrito a cada
       borda de subida do
22   -- clock, desde que PC_we = '1'. O sinal PC_we pode ser colocado em nível '0'
       devido a stalls no pipeline.
23   write_PC: process (rst , clk , PC_we)
24   begin
25     if (rising_edge(clk)) then
26       if (rst = '1') then
27         PC <= (others => '0');
28
29       elsif (PC_we = '1') then
30         PC <= PC_next;
31
32       end if;
33     end if;
34   end process;
35
36 end architecture behavior;
```

F.12 – REGFILE.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.core_pkg.all;
5
6  -- @file RegFile.vhd
7  -- @author Kevin Morais (moraisku@gmail)
8  -- @date 2018
9  -- @version 1
10 -- Possui 16 registradores de 4 bytes, x0 a x15, sendo o registrador x0 o de
      valor 0 fixo, x1 o link na convenção do RISC-V,
11 -- que armazena o endereço de PC em instr. de desvio (jump and link). Possui
      duas saídas de dados, data_rs1 e data_rs2.
12 -- A escrita érealizada no endereço dado por rd quando se tem a permissão de
      escrita RegFile_W = '1' e uma borda de subida
```

```
13 -- do clock, o dado a ser escrito corresponde a data_rd. O Banco de
       Registradores realizar internamente a técnica de Forward,
14 -- por meio da comparação do endereço de escrita 'rd' com alguma das entradas de
       endereço rs1 e/ou rs2. Se forem iguais,
15 -- transfere data_rd para a saída correspondente no lugar do valor que
       supostamente seria lido no Banco de Registradores,
16 -- isso apenas se RegFile_W = '1'.
17 entity RegFile is
18     port(clk : in std_logic;       -- sinal de clock
19          rst  : in std_logic;      -- sinal de reset
20     RegFile_W : in std_logic;       -- sinal de permissão de escrita no banco de
           registradores
21     rs1 : in std_logic_vector(3 downto 0); -- endereço de leitura rs1 do banco
           de registradores
22     rs2  : in std_logic_vector(3 downto 0); -- endereço de leitura rs2 do banco
           de registradores
23     rd  : in std_logic_vector(3 downto 0); -- endereço de escrita do banco de
           registradores
24     data_rd : in std_logic_vector(n-1 downto 0);  -- dado a ser escrito no banco
            de registradores
25     data_rs1 : out std_logic_vector(n-1 downto 0); -- saída de dados do endereço
            rs1
26     data_rs2 : out std_logic_vector(n-1 downto 0));-- saída de dados do endereço
            rs2
27 end entity RegFile;
28
29 architecture behavior of RegFile is
30
31   -- type array, 16x4bytes, banco de registradores
32   type reg_array is ARRAY(0 to n_E-1) of std_logic_vector(n-1 downto 0);
33   signal Register_File : reg_array;
34
35 begin
36
37   data_rs1 <= data_rd when (rs1 = rd AND RegFile_W = '1') else
38         Register_File(conv_integer(rs1));
39
40   data_rs2 <= data_rd when (rs2 = rd AND RegFile_W = '1') else
41         Register_File(conv_integer(rs2));
42
43   -- Processo de escrita no Banco de Registradores
```

```
44   -- Caso o sinal reset esteja em nível '1', s os regs. são resetados na borda
         de subida
45   -- de clk, do contrário, se RegFile_W = '1' ocorre a escrita de data_rd no
         endereço rd,
46   -- com exceção do endereço 0, que éignorado.
47   RegFile_Write : process (clk, rst, RegFile_W)
48   begin
49     if (rising_edge(clk)) then
50       if (rst = '1') then
51         Register_File <= (others => (others => '0'));
52
53       elsif (RegFile_W = '1' AND rd /= "0000") then
54         Register_File(conv_integer(rd)) <= data_rd;
55
56       end if;
57     end if;
58   end process RegFile_Write;
59
60 end architecture behavior;
```

F.13 – STAGE1_IF.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.core_pkg.all;
5
6  -- @file stage1_IF.vhd
7  -- @author Kevin Morais (moraisku@gmail)
8  -- @date 2018
9  -- @version 1
10 -- @brief Instruction Fetch (IF)
11 -- @details Primeiro estágio de execução do pipeline, responsável em geral pela
12 -- busca da instrução da memória. A fonte de endereço de acesso da memória pode
13 -- ser do ProgramCounter, Preditor de Desvios (prevendo taken = '1'), ou então
14 -- do endereço alvo cálculado no estágio 4 (MEM) do pipeline. Também épossível
15 -- obter um caso a mais, em que o preditor tenha errado ao utilizar taken = '1',
16 -- e o valor deve ser corrigido para o correto, que também éprovido pelo estágio
17 -- 4 (MEM).
18 entity stage1_IF is
```

```
19      port(rst   : in std_logic;  -- sinal de reset
20      clk    : in std_logic;  -- sinal de clk
21      wait_data : in std_logic;  -- sinal que indica dado pronto da memória
22          wait_instr : in std_logic;  -- sinal que indica que a instr. da memória
                está pronta
23          Mem_RW : in std_logic;  -- sinal de leitura/escrita da memória
24      rvc : in std_logic;  -- sinal que indica que a instr. buscada éda ext C (RVC
            )
25      -- sinais de stall do pipeline, e registrador de pipeline IF/ID
26      predict_flush : out std_logic; -- sinal de flush por miss predict
27      stall_IF_ID  : in std_logic;  -- sinal de stall do estágio IF, ativo em '1',
             passa uma instrução 'NOP' para IF/ID
28      stall_ID_IF  : in std_logic;  -- sinal de stall do estágio IF, ativo em '1',
             desabilita a escrita de IF/ID
29      IF_ID   : out std_logic_vector(n_IF_ID-1 downto 0); --! registrador de
            pipeline IF/ID
30      -- sinais de controle
31      Branch   : in std_logic;  -- sinal de controle que indica que a fonte de
            desvio écondicional (branch), utilizado na BHT
32      Branch_unc  : in std_logic;  -- sinal que indica um desvio incond. (jump),
            utilizado para gerar o stall
33      PC_Src   : in std_logic;  -- sinal de controle que controla o MUX 0 do data
            path, corresponde ao LSb
34      -- endereço alvo e o valor de PC no estágio MEM
35      Branch_addr_MEM : in std_logiC_vector(n-1 downto 0); -- endereço de branch
            provido pelo estágio Memory
36      PC_MEM   : in std_logic_vector(n-1 downto 0); -- endereço do PC no estágio
            MEM, utilizado na predict unit
37      PC_MEM_offset  : in std_logic_vector(n-1 downto 0); -- endereço do PC +
            offset (2 ou 4) no estágio MEM, utilizado para correção da ação do
            preditor
38      -- instrução buscada, e o valor de PC no estágio IF
39      instr_IF  : in std_logic_vector(n_I-1 downto 0); -- entrada de dados de
            programa vindo da memória
40      PC_IF   : out std_logic_vector(n-1 downto 0)); -- endereço contido no PC
41  end entity stage1_IF;
42
43  architecture behavior of stage1_IF is
44
45    -- codificação da instr. NOP, pelos 3 LSB, utilizado para inserir NOPs no está
          gio ID
```

```
46    constant NOP_instr : std_logic_vector(2 downto 0) := "100";
47
48 -- Sinais
49   -- Program Counter
50   signal PC_next : std_logic_vector(n-1 downto 0); -- entrada do PC
51   signal PC  : std_logic_vector(n-1 downto 0); -- saída do PC
52   signal PC_4  : std_logic_vector(n-1 downto 0); -- PC+4 (ou +2 para instr. RVC)
53   signal PC_we : std_logic;      -- habilitação de escrita do PC
54
55   -- Predic Unit
56   signal taken : std_logic;   -- sinal que preve o branch como taken, quando em
         '1'
57   signal miss_predict: std_logic;      -- sinal que indica que ocorreu miss
        predict
58   signal miss_check : std_logic;      -- sinal que verifica se deu miss predict
        como taken ou not taken
59   signal addr_target : std_logic_vector(n-1 downto 0);-- endereço alvo da previs
        ão taken
60
61   signal MO_out : std_logic_vector(n-1 downto 0);
62
63 begin
64   -- endereço de PC, utilizado para buscar instruções
65    PC_IF <= PC;
66
67    -- sinal de Stall para desvios incondicionais e condicionais em que o
         preditor errou (miss)
68   predict_flush <= miss_predict OR Branch_unc;
69
70   -- Program Counter write enable
71   PC_we <= (NOT(stall_IF_ID OR stall_ID_IF) OR miss_predict OR Branch_unc) AND
        NOT(wait_data OR wait_instr);
72
73    PC_4 <= PC + 4 when (rvc = '0') else
74            PC + 2; -- when (rvc = '1');
75
76   -- MUX 0 : define o próximo valor a ser escrito no PC, entre PC+4 e algum
        desvio gerado,
77   -- possui também o endereço para correção do valor de PC (caso o preditor erre
        )
78   PC_next <= addr_target when (taken = '1') else
```

```
79        M0_out;
80
81    M0_out <= PC_MEM_offset when (miss_predict = '1' AND miss_check = '1') else
82          Branch_addr_MEM when (PC_Src = '1') else
83          PC_4    when (PC_Src = '0');
84
85    -- Processo de escrita no registrador IF/ID.
86    -- Caso rst = '1' ou stall = '1', passa NOP para o registrador, do contrário
87    -- armazena a instr. atual. Tanto o rst quanto o stall são síncronos.
88    process(clk, rst, stall_IF_ID, stall_ID_IF)
89    begin
90      if (rising_edge(clk)) then
91        if (rst = '1') then
92          IF_ID(n_IF_ID-1 downto 3) <= (others => '0');
93          IF_ID(2 downto 0) <= NOP_instr; -- inseri um NOP o estágio seguinte (ID)
94
95        -- prioriedade maior com 'wait_data' = 'stall_ID_IF', seguido por '
              wait_instr' = 'stall_IF_ID'
96            elsif (wait_data = '1' OR stall_ID_IF = '1') then
97                -- não escreve
98
99            elsif (stall_IF_ID = '1' OR wait_instr = '1') then
100               IF_ID(n_IF_ID-1 downto 3) <= (others => '0');
101               IF_ID(2 downto 0) <= NOP_instr;
102
103       else -- (stall_IF_ID = '0') then
104               IF_ID(n_IF_ID-1) <= rvc;
105               IF_ID(n_IF_ID-2 downto n_I) <= PC(n-1 downto 0);
106         IF_ID(n_I-1 downto 0) <= instr_IF;
107
108       end if;
109     end if;
110   end process;
111 --------------------------
112 -- Instance
113 --------------------------
114   -- Program Counter instance
115   ProgramCounter_i: ProgramCounter
116   port map
117   (rst => rst ,
118   clk => clk ,
```

```
119   PC_we => PC_we ,
120   PC => PC ,
121   PC_next => PC_next
122   );
123
124   -- Prediction_unit instace
125   Predict_unit_i: Predict_unit
126   port map
127   (clk => clk ,
128   rst => rst ,
129     wait_data => wait_data ,
130     wait_instr => wait_instr ,
131     Mem_RW => Mem_RW ,
132   Branch => Branch ,
133   PC_Src => PC_Src ,
134   Branch_addr_MEM => Branch_addr_MEM ,
135   miss_predict => miss_predict ,
136   taken => taken ,
137   miss_check => miss_check ,
138   addr_target => addr_target ,
139   -- PC_next => PC_next(n-1 downto 0+b_offset) ,
140   PC_next   => M0_out(n-1 downto 0+b_offset) ,
141   PC_MEM => PC_MEM(n-1 downto 0+b_offset)
142   );
143
144 end architecture behavior;
```

F.14 – STAGE2_ID.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.core_pkg.all;
4
5  -- @file stage2_ID.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2018
8  -- @version 1
9  -- Instruction Decode (ID).
10 -- Realiza a decodificação da instrução da base I do RISC-V, pois a base E
      implementa o mesma ISA base.
```

```
11 -- O campo opcode corresponde aos campos [6:2] da instrução, sendo os 2 msb
       descartados no estágio 1 de
12 -- busca da instrução (IF), pois não adicionam mais informação útil ao
       processador após a decodificação
13 -- do comprimento da instrução. A decodificação do campo imediato da instrução
       ocorre em paralelo a dos
14 -- sinais de controle (que ocorre em Control_unit.vhd), sendo a unidade responsá
       vel a Imm_unit.vhd.
15 -- O banco de registradores possui um total de 16 registradores de uso geral,
       conforme o proposto na
16 -- base E do RISC-V, com duas portas de leitura assíncrona, rs1 e rs2 de 4 bits,
        e suas respectivas saídas
17 -- de 32 bits, data_rs1 e data_rs2. Também conta com uma porta de entrada para
       endereço de escrita, rd, e
18 -- entrada de dado a ser escrito, data_rd. A unidade de detecção de hazards de
       dados, devido a instruçóes
19 -- de LW seguidas por aritméticas/lógicas também esta localizada nesse estágio,
       atuando de forma a inserir
20 -- NOPs no pipeline quando necessário.
21 entity stage2_ID is
22   port(rst  : in std_logic; -- sinal de reset
23     clk   : in std_logic; -- sinal de clk
24     wait_data : in std_logic; -- sinal que indica dado pronto da memória
25     -- sinais de controle
26     RegFile_W : in std_logic; -- sinal de controle RegFile_W
27     Mem_R_EX : in std_logic; -- sinal de controle Mem_R no estágio EX
28     -- sinais do banco de regs.
29     rd_EX  : in std_logic_vector(3 downto 0); -- endereço destino de escrita do
           banco de regs. no estágio EX
30     rd_WB  : in std_logic_vector(3 downto 0); -- endereço destino de escrita do
           banco de regs. no estágio WB
31     data_rd_WB : in std_logic_vector(n-1 downto 0);-- dado de entrada do banco
           de regs.
32     -- pipeline
33     IF_ID  : in std_logic_vector(n_IF_ID-1 downto 0);  -- registrador de entrada
            do estágio Instr_Decode, regs. entre IF e ID
34     flush_ID_EX : in std_logic;        -- sinal de flush do reg. de pipelin ID/
           EX, no caso de miss_predict de branch
35     stall_ID_IF : out std_logic;       -- sinal de stall do reg. de pipeline ID/
           IF
36     ID_EX  : out std_logic_vector(n_ID_EX-1 downto 0));-- registrador de saída
```

```
       do estágio Instr_Decode, regs. entre ID e EX
end entity stage2_ID;

architecture behavior of stage2_ID is

  -- identificação dos campos do registrador IF/ID
  alias rvc_ID : std_logic      is IF_ID(n+n_I);    -- campo que possui o sinal
      rvc
  alias PC_ID  : std_logic_vector(n-1 downto 0) is IF_ID(n+n_I-1 downto n_I); --
       campo que possui o valor de PC
  alias instr_ID : std_logic_vector(n_I-1 downto 0) is IF_ID(n_I-1 downto 0); --
       campo que possui a instr.

  -- identificação dos campos do registrador ID/EX
  alias rvc_EX   : std_logic      is ID_EX(154);    -- campo que possui o sinal
      rvc, indica se o incremento de PC foi 2 ou 4
  alias ctrl_ID_EX : std_logic_vector(10 downto 0) is ID_EX(153 downto 143); --
      campo que possui os sinais de controle dos próximos estágios
  alias PC_EX  : std_logic_vector(n-1 downto 0) is ID_EX(142 downto 111);  --
      campo que possui o valor do PC
  alias data_rs1_EX : std_logic_vector(n-1 downto 0) is ID_EX(110 downto 79);
      -- campo do dado lido de rs1 do banco de regs.
  alias data_rs2_EX : std_logic_vector(n-1 downto 0) is ID_EX(78 downto 47);  --
       campo do dado lido de rs2 do banco de regs.
  alias imm_EX : std_logic_vector(n-1 downto 0) is ID_EX(46 downto 15); -- campo
       que possui o sinal imediato já extendido
  alias funct3 : std_logic_vector(2 downto 0) is ID_EX(14 downto 12); -- campo
      que possui os bits de decodificação do subcontrole da ALU
  alias rs1_EX  : std_logic_vector(3 downto 0) is ID_EX(11 downto 8);  -- campo
      que possui o addr rs1
  alias rs2_EX  : std_logic_vector(3 downto 0) is ID_EX(7 downto 4);  -- campo
      que possui o addr rs2
  alias rd_EX_t : std_logic_vector(3 downto 0) is ID_EX(3 downto 0); -- campo
      que possui o addr rd

  -- campos da instrução sinais de decodificação da instrução
  signal imm_field : std_logic_vector(31 downto 7);
  signal rs1_ID : std_logic_vector(3 downto 0);
  signal rs2_ID : std_logic_vector(3 downto 0);
  signal rd_ID : std_logic_vector(3 downto 0);
  signal funct3_ID : std_logic_vector(2 downto 0);
```

```
64    signal opcode : std_logic_vector(6 downto 2);
65
66    -- sinais de Controle_unit
67    signal ctrl_ID    : std_logic_vector(10 downto 0);
68    alias RegFile_W_ID_temp : std_logic is ctrl_ID(10);
69    signal RegFile_W_ID  : std_logic;
70
71    -- sinais do RegFile
72    signal data_rs1_ID : std_logic_vector(n-1 downto 0);
73    signal data_rs2_ID : std_logic_vector(n-1 downto 0);
74
75    -- sinais de Imm_unit
76    signal imm_out : std_logic_vector(n-1 downto 0);
77
78    -- sinais de Hazards_unit
79    signal stall_ID_EX  : std_logic;
80    signal Mem_W_ID  : std_logic;
81
82  begin
83
84      Mem_W_ID <= ctrl_ID(4);
85
86      -- sinal de stall para os estágios anteriores, devido a detecção de hazards
           por 'Hazards_unit.vhd'
87    stall_ID_IF <= stall_ID_EX;
88
89      -- conexão dos sinais dos campos da instrução ao reg. de pipeline IF/ID
90      imm_field <= instr_ID(n_I-1 downto 7-length_dec);
91      rs1_ID <= instr_ID(18-length_dec downto 15-length_dec);
92      rs2_ID <= instr_ID(23-length_dec downto 20-length_dec);
93      rd_ID <= instr_ID(10-length_dec downto 7-length_dec);
94      funct3_ID <= instr_ID(14-length_dec downto 12-length_dec);
95      opcode <= instr_ID(6-length_dec downto 2-length_dec);
96
97    -- desabilita a escrita (ignora) se o regs. destino for x0
98    RegFile_W_ID <= '0' when (rd_ID = "0000") else
99           RegFile_W_ID_temp;
100
101   -- Processo de escrita no registrador ID/EX. Caso rst = '1' ou stall = '1',
102   -- passa NOP para o registrador.
103   process (rst, clk, stall_ID_EX , flush_ID_EX)
```

```
104  begin
105    if (rising_edge(clk)) then
106      if (rst = '1') then
107        ID_EX <= (others => '0');
108
109      -- prioriedade maior sobre o sinal 'wait_data', e depois sobre 'stall_ID_EX
             ' = 'flush_ID_EX' de mesmo nível
110      elsif (wait_data = '1') then
111        -- não escreve
112
113      elsif (stall_ID_EX = '0' AND flush_ID_EX = '0') then
114        rvc_EX   <= rvc_ID;
115        ctrl_ID_EX(10)    <= RegFile_W_ID;
116        ctrl_ID_EX(9 downto 0) <= ctrl_ID(9 downto 0);
117        PC_EX <= PC_ID;
118        data_rs1_EX <= data_rs1_ID;
119        data_rs2_EX <= data_rs2_ID;
120        imm_EX  <= imm_out;
121        funct3  <= funct3_ID;
122        rs1_EX  <= rs1_ID;
123        rs2_EX  <= rs2_ID;
124        rd_EX_t <= rd_ID;
125
126      else -- stall_ID_EX = '1' OR flush_ID_EX = '1', inseri um NOP
127        ctrl_ID_EX <= (others => '0'); -- flush por miss branch ou data hazard (
               load seguido de aritim/logic)
128
129      end if;
130    end if;
131  end process;
132
133    -------------------------------------------------------
134  -- instanciação do Banco de Registradores
135  RegFile_i: RegFile
136  port map
137  (rst => rst ,
138  clk => clk ,
139  RegFile_W => RegFile_W ,
140  rs1 => rs1_ID ,
141  rs2 => rs2_ID ,
142  rd => rd_WB ,
```

```
143   data_rs1 => data_rs1_ID ,
144   data_rs2 => data_rs2_ID ,
145   data_rd => data_rd_WB
146   );
147
148   -- instanciação da Unidade de Imediato
149   Imm_unit_i: Imm_unit
150   port map
151   (opcode => opcode ,
152   imm_field => imm_field ,
153   imm_out => imm_out
154   );
155
156   -- instanciação da Unidade de Controle
157   Control_i: Control
158   port map
159   (opcode => opcode ,
160   ctrl_ID => ctrl_ID
161   );
162
163   -- instanciação da Unidade de Hazards
164   Hazards_unit_i: Hazards_unit
165   port map
166   (rs1_ID => rs1_ID ,
167   rs2_ID => rs2_ID ,
168   rd_EX => rd_EX ,
169   Mem_R_EX => Mem_R_EX ,
170   Mem_W_ID => Mem_W_ID ,
171   stall_ID_EX => stall_ID_EX
172   );
173
174 end architecture behavior;
```

F.15 – STAGE3_EX.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use work.core_pkg.all;
5
```

```
6  -- @file stage3_EX.vhd
7  -- @author Kevin Morais (moraisku@gmail)
8  -- @date 2018
9  -- @version 1
10 -- Execute (EX), executa as operações lógicas/aritméticas da instrução, como cá
       lculo de endereços desvio
11 -- condicional/incodicional, endereços de acesso a memória, comparações, somas,
       entre outros.
12 entity stage3_EX is
13   port(rst : in std_logic; -- sinal de reset
14     clk : in std_logic; -- sinal de clk
15     wait_data : in std_logic; -- sinal que indica dado pronto da memória
16     -- foward
17     Forward_A : in std_logic_vector(1 downto 0); -- sinal que controla o MUX A
           do data path, Forwarding
18     Forward_B : in std_logic_vector(1 downto 0); -- sinal que controla o MUX B
           do data path, Forwarding
19     Forward_S_EX : in std_logic;     -- sinal que controla o MUX S_EX do data
           path, Forwarding
20     fw_data_MEM : in std_logic_vector(n-1 downto 0);-- sinal a ser escrito no
           banco de regs., no estagio MEM
21     fw_data_WB : in std_logic_vector(n-1 downto 0);-- sinal a ser escrito no
           banco de regs., no estagio WB
22     -- pipeline
23     flush_EX_MEM : in std_logic;       -- sinal de flush do estagio xecute,
           ativo em '1'
24     ID_EX : in std_logic_vector(n_ID_EX-1 downto 0); -- registrador de entrada
           do estagio Execute (entre ID e EX)
25     EX_MEM  : out std_logic_vector(n_EX_MEM-1 downto 0));-- registrador de saída
            do estagio Execute (entre EX e MEM)
26 end entity stage3_EX;
27
28 architecture behavior of stage3_EX is
29
30   -- identificao dos campos do registrador ID/EX
31   alias rvc_EX   : std_logic      is ID_EX(154);    -- campo que possui o sinal
         rvc_EX, indicando se o incremento de PC foi 2 ou 4
32   alias ctrl_ID_EX : std_logic_vector(7 downto 0) is ID_EX(153 downto 146);  --
         campo que possui os sinais de controle dos estagios MEM e WB
33   alias ALU_op  : std_logic_vector is ID_EX(145 downto 144); -- campo que possui
        o sinal de controle ALU_op
```

```
34  alias ALU_oprB : std_logic      is ID_EX(143);    -- campo que possui o sinal
        de controle ALU_oprB
35  alias PC_EX  : std_logic_vector(n-1 downto 0) is ID_EX(142 downto 111);  --
        campo que possui o valor do PC
36  alias data_rs1_EX : std_logic_vector(n-1 downto 0) is ID_EX(110 downto 79);
        -- campo do dado lido de rs1 do banco de regs.
37  alias data_rs2_EX : std_logic_vector(n-1 downto 0) is ID_EX(78 downto 47);  --
         campo do dado lido de rs2 do banco de regs.
38  alias imm_EX : std_logic_vector(n-1 downto 0) is ID_EX(46 downto 15); -- campo
         que possui o sinal imediato a ser extendido no estágio EX
39  alias funct3 : std_logic_vector(2 downto 0) is ID_EX(14 downto 12); -- campo
        que possui os bits de decodificação do subcontrole da ALU
40  alias rs1_EX  : std_logic_vector(3 downto 0) is ID_EX(11 downto 8);  -- campo
        que possui o addr rs1
41  alias rs2_EX  : std_logic_vector(3 downto 0) is ID_EX(7 downto 4);  -- campo
        que possui o addr rs2
42  alias rd_EX : std_logic_vector(3 downto 0) is ID_EX(3 downto 0); -- campo que
        possui o addr rd
43
44  -- identificao dos campos do registrador EX/MEM
45  alias rvc_MEM  : std_logic      is EX_MEM(148);   -- campo que possui o sinal
        rvc no estágio MEM
46  alias ctrl_EX_MEM : std_logic_vector(7 downto 0) is EX_MEM(147 downto 140);--
        campo que possui os sinais de controle dos estagios MEM e WB
47  alias PC_MEM  : std_logic_vector(n-1 downto 0) is EX_MEM(139 downto 108);--
        campo que possui o valor do PC
48  alias PC_Imm_MEM : std_logic_vector(n-1 downto 0) is EX_MEM(107 downto 76); --
         campo que possui o endereço alvo do desvio
49  alias ALU_result_MEM: std_logic_vector(n-1 downto 0) is EX_MEM(75 downto 44);
        -- campo que possui o resultado da ALU
50  alias Zero_MEM  : std_logic     is EX_MEM(43);   -- campo que possui o sinal
        de controle Zero para instr. de desvio
51  alias funct3_MEM : std_logic_vector(2 downto 0) is EX_MEM(42 downto 40); --
        campo funct3 no estágio MEM
52  alias data_rs2_MEM : std_logic_vector(n-1 downto 0) is EX_MEM(39 downto 8); --
         campo que possui o dado de saída rs2 no estagio MEM
53  alias rs2_MEM  : std_logic_vector(3 downto 0) is EX_MEM(7 downto 4);  -- campo
         que possui o endereço de rs2 no estagio MEM
54  alias rd_MEM  : std_logic_vector(3 downto 0) is EX_MEM(3 downto 0);  -- campo
        que possui o reg. destino do banco de regs.
55
```

```
56      -- sinal PC+Imm
57      signal PC_Imm_EX : std_logic_vector(n-1 downto 0);
58
59    -- sinais da ALU
60    signal opr_A : std_logic_vector(n-1 downto 0); -- operando A da ALU
61    signal opr_B : std_logic_vector(n-1 downto 0); -- operando B da ALU
62    signal ALU_out : std_logic_vector(n-1 downto 0); -- saída da ALU
63    signal ALU_ctrl : std_logic_vector(3 downto 0); -- sinal de controle de
          entrada da ALU
64    signal ALU_zero : std_logic; -- sinal de verificação de igualdade a 0 do
          resultado da operação na ALU
65    signal fct : std_logic; -- sinal do bit [30] da instrução, que corresponde ao
          [10] do imediato nas instruções que utilizam o campo funct3 para decodifica
          ção da ALU
66
67      -- sinais de saída dos multiplexadores
68    signal MA_out : std_logic_vector(n-1 downto 0); -- saída do MUX A do data path
          , no estagio EX
69    signal MB_out : std_logic_vector(n-1 downto 0); -- saída do MUX B do data path
          , no estagio EX
70    signal MS_EX : std_logic_vector(n-1 downto 0); -- saída do MUX S EX do data
          path, no estágio EX
71
72  begin
73
74    -- PC + immediate
75    PC_Imm_EX <= PC_EX + imm_EX(n-1 downto 0);
76
77    ----- MUX A, define o operando_A da ALU
78      MA_out <= data_rs1_EX when (Forward_A = "00") else
79                fw_data_WB when (Forward_A = "01") else
80                fw_data_MEM; -- when (Forward_A = "10")
81
82    opr_A <= MA_out;
83
84    ----- MUX 2 seguido pelo MUX B, definem o operando_B da ALU
85    MB_out <= data_rs2_EX when (Forward_B = "00") else
86              fw_data_WB when (Forward_B = "01") else
87          fw_data_MEM; -- when (Forward_B = "10");
88
89    opr_B <= MB_out when (ALU_oprB = '0') else
```

```
90       imm_EX; -- when (ALU_oprB = '1');
91
92    ----- MUX S EX, define o dado a ser escrito na memória
93    MS_EX <=  data_rs2_EX when (Forward_S_EX = '0') else
94              fw_data_WB; -- when (Forward_S_EX = '1');
95
96    -- Subunidade de Controle da ALU
97    fct <= imm_EX(10);
98
99    ALU_ctrl <= "1010" when (ALU_op = "11") else -- LUI
100                "0000" when (ALU_op = "00") else -- ADD
101                "0001" when ((funct3 = "000" AND fct = '1' AND ALU_oprB = '0') OR
                       (ALU_op = "10" AND funct3(2 downto 1) = "00")) else -- SUB /
102         "1000" when (funct3 = "010" OR (ALU_op = "10" AND funct3(1) = '0')) else
              -- STL
103         "1001" when (funct3 = "011" OR (ALU_op = "10" AND funct3(1) = '1')) else
              -- STLU
104         "0010" when (funct3 = "001") else -- SLL
105                "0011" when (funct3 = "101" AND fct = '0') else -- SRL
106                "0100" when (funct3 = "100") else -- XOR
107                "0101" when (funct3 = "101" AND fct = '1') else -- SRA
108                "0110" when (funct3 = "110") else -- OR
109                "0111" when (funct3 = "111") else -- AND
110                "0000"; -- ADD
111
112   -- Processo de escrita no registrador EX/MEM. Caso rst = '1' ou flush = '1',
113   -- passa NOP para o registrador.
114   process (rst, clk, flush_EX_MEM)
115   begin
116     if (rising_edge(clk)) then
117       if (rst = '1') then
118         EX_MEM <= (others => '0');
119
120       -- prioriedade maior sobre o sinal 'wait_data', e depois sobre o '
              flush_EX_MEM'
121       elsif (wait_data = '1') then
122         -- não escreve
123
124       elsif (flush_EX_MEM = '1') then -- flush por miss branch
125         ctrl_EX_MEM <= (others => '0'); -- NOP
126
```

```
127       else
128       --elsif (flush_EX_MEM = '0' AND wait_data = '0') then
129         rvc_MEM   <= rvc_EX;
130         ctrl_EX_MEM <= ctrl_ID_EX ;
131         PC_MEM   <= PC_EX;
132         PC_Imm_MEM  <= PC_Imm_EX;
133         ALU_result_MEM <= ALU_out;
134         Zero_MEM  <= ALU_zero;
135         funct3_MEM <= funct3;
136         data_rs2_MEM  <= MS_EX;
137         rs2_MEM   <= rs2_EX;
138         rd_MEM   <= rd_EX;
139
140       end if;
141     end if;
142   end process;
143
144   -- instanciao da ALU
145   ALU_i: ALU
146   port map
147   (opr_A => opr_A ,
148   opr_B => opr_B ,
149   ALU_out => ALU_out ,
150   ALU_ctrl => ALU_ctrl ,
151     ALU_zero => ALU_zero
152   );
153
154 end architecture behavior;
```

F.16 – STAGE4_MEM.VHD

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 use work.core_pkg.all;
5
6 -- @file stage4_MEM.vhd
7 -- @author Kevin Morais (moraisku@gmail)
8 -- @date 2018
9 -- @version 1
```

```
10 -- Memory (MEM), realiza o acesso a memória, para escrita ou leitura de dados. Possuí
11 -- duas subunidades de controle, uma responsável por definir entre o tipo de Store:
12 -- Store Word, Store Half Word ou Store Byte. Outra subunidade para identificar o tipo
13 -- de instrução branch em execução: BEQ, BNE, BGE[U] ou BLT[U]. Realiza também o
14 -- incremento de PC novamente, de acordo com o sinal 'rvc', para enviar ao está gio 1
15 -- (IF) e realizar a lógica de controle do endereço atual de PC.
16
17 entity stage4_MEM is
18   port( rst    : in std_logic;         -- sinal de reset
19       clk    : in std_logic;         -- sinal de clk
20       wait_data  : in std_logic;         -- sinal que indica dado pronto da memó ria
21       EX_MEM   : in std_logic_vector(n_EX_MEM-1 downto 0); -- registrador de entrada do estagio Memory (entre EX e MEM)
22       Forward_S  : in std_logic;         -- sinal que controla o MUX S do data path
23       fw_data_WB  : in std_logic_vector(n-1 downto 0);  -- sinal a ser escrito no banco de regs., no estagio WB, forwarding
24       PC_Src   : out std_logic;        -- sinal de controle PC_Src
25       PC_MEM_offset : out std_logic_vector(n-1 downto 0);  -- sinal de PC após o incremento por 4 (ou 2 se rvc = '1')
26       Branch_addr_MEM : out std_logic_vector(n-1 downto 0);  -- sinal com o endereço alvo de branch (condicional e incondicional)
27       MEM_WB   : out std_logic_vector(n_MEM_WB-1 downto 0);-- registrador de saí da do estagio Memory (entre MEM e WB)
28       sb_en    : out std_logic;        -- store byte enable
29       sh_en    : out std_logic;        -- store half enable
30       data_rd_MEM  : out std_logic_vector(n-1 downto 0);  -- sinal a ser escrito no banco de regs., no estagio MEM
31       data_BUS   : inout std_logic_vector(n-1 downto 0)); -- barramento de dados interno do processador
32 end entity stage4_MEM;
33
34 architecture behavior of stage4_MEM is
35
36   -- identificacao dos campos do registrador EX/MEM
37   alias rvc_MEM   : std_logic      is EX_MEM(148);   -- campo que possui o sinal
```

```
     de controle ruc no estágio MEM
38   alias ctrl_EX_MEM : std_logic is EX_MEM(147); -- campo que possui os sinais de
     controle do estagio Write Back
39   alias RegFile_Src_0 : std_logic_vector(1 downto 0) is EX_MEM(146 downto 145);
     -- campo que possui o sinal de controle RegFile_Src_0
40   alias Branch_unc  : std_logic      is EX_MEM(144);   -- campo que possui o
     sinal de controle Branch_unc
41   alias Branch   : std_logic      is EX_MEM(143);   -- campo que possui o sinal
     de controle Branch
42   alias Mem_R    : std_logic      is EX_MEM(142);   -- campo que possui o sinal
     de controle Mem_R
43   alias Mem_W    : std_logic      is EX_MEM(141);   -- campo que possui o sinal
     de controle Mem_W
44   alias Branch_Src  : std_logic      is EX_MEM(140);   -- campo que possui o
     sinal de controle Branch_Src
45   alias PC_MEM   : std_logic_vector(n-1 downto 0) is EX_MEM(139 downto 108); --
     campo que possui o valor de PC
46   alias PC_Imm_MEM  : std_logic_vector(n-1 downto 0) is EX_MEM(107 downto 76);
     -- campo que possui o valor de desvio do branch (condicional e incondiconal
     )
47   alias ALU_result_MEM : std_logic_vector(n-1 downto 0) is EX_MEM(75 downto 44);
      -- campo que possui o end. de acesso da memoria/dado a ser escrito no
     Banco de Regs.
48    alias Zero_MEM   : std_logic      is EX_MEM(43);   -- campo que possui o
      sinal de controle Zero para instr. de desvio
49   alias funct3_MEM : std_logic_vector(2 downto 0) is EX_MEM(42 downto 40); --
     campo funct3 da instrução, no estágio MEM
50   alias data_rs2_MEM : std_logic_vector(n-1 downto 0) is EX_MEM(39 downto 8); --
     campo que possui o dado a ser escrito na memória/banco de regs.
51   alias rs2_MEM   : std_logic_vector(3 downto 0) is EX_MEM(7 downto 4); -- campo
     que possui o endereço de leitura rs2 do Banco de Regs.
52   alias rd_MEM   : std_logic_vector(3 downto 0) is EX_MEM(3 downto 0);  -- campo
     que possui o endereço de escrita do Banco de Regs.
53
54   -- identificacao dos campos do registrador MEM/WB
55   alias ctrl_MEM_WB : std_logic_vector(1 downto 0) is MEM_WB(72 downto 71); --
     campo que possui os sinais de controle do estagio Write Back
56   alias data_WB_0 : std_logic_vector(n-1 downto 0) is MEM_WB(70 downto 39); --
     campo que possui o dado 0 a ser escrito no banco de regs.
57   alias data_WB_1  : std_logic_vector(n-1 downto 0) is MEM_WB(38 downto 7); --
     campo que possui o dado 1 a ser escrito no banco de regs.
```

```
58   alias funct3_WB : std_logic_vector(2 downto 0) is MEM_WB(6 downto 4); -- campo
         funct3 noe stágio WB
59   alias rd_WB   : std_logic_vector(3 downto 0) is MEM_WB(3 downto 0); -- campo
        que possui o endereço de escrita do Banco de Regs.
60
61     -- sinais de Branch Decode
62     signal zero_true : std_logic;
63     signal slt_true : std_logic;
64     signal B_true : std_logic;
65
66   -- MUX 4 e S do datapath
67   signal PC_offset: std_logic_vector(n-1 downto 0); -- sinal de PC + offset (que
         pode ser 2 ou 4, de acordo com o sinal ruc)
68     signal M4_out  : std_logic_vector(n-1 downto 0); -- sinal de saída do MUX 4
          do data path
69   signal MS_out : std_logic_vector(n-1 downto 0);  -- sinal de saída do MUX S do
         data path
70
71 begin
72   -- sinais sb_en e sh_en (store byte enable e store half enable)
73   -- e load byte half enable
74   sb_en <= '1' when (funct3_MEM(1) = '0' AND funct3_MEM(0) = '0') else '0';
75   sh_en <= '1' when (funct3_MEM(1) = '0' AND funct3_MEM(0) = '1') else '0';
76
77     -- Branch Decode, identifica qual sinal de branch éválido, para BGE(U), BLT(
          U) e BEQ(NE)
78     zero_true <= (Zero_MEM XOR funct3_MEM(0) ) AND NOT(funct3_MEM(2));
79     slt_true <= (ALU_result_MEM(0) XOR funct3_MEM(0) ) AND funct3_MEM(2);
80     B_true <= zero_true OR slt_true;
81
82   -- Lógica do sinal de controle PC_Src, que controla o MUX 0 (M0) do datapath
83   PC_Src <= Branch_unc OR (B_true AND Branch);
84
85   -- Lógica do uso do barramento de dados interno do estágio MEM
86   data_BUS <= MS_out when (Mem_W = '1') else  -- barramento de dados do estágio
         MEM
87         (others => 'Z');
88
89   -- MUX 3 : define o endereço de desvio, entre PC+Imm (PC_Imm_MEM) e a saída da
         ALU (ALU_result_MEM)
90   Branch_addr_MEM <= PC_Imm_MEM when (Branch_Src = '0') else
```

```
91            ALU_result_MEM; -- when (Branch_Src = '1');
92
93  -- MUX 4 : define a entrada de dados do Banco de Regs., entre PC e a saída da
        ALU (ALU_result_MEM)
94  PC_offset <= PC_MEM + 2 when (rvc_MEM = '1') else
95        PC_MEM + 4; -- when (rvc_MEM = '0');
96  PC_MEM_offset <= PC_offset;
97
98  M4_out <= PC_offset when (RegFile_Src_0 = "00") else
99            PC_Imm_MEM when (RegFile_Src_0 = "01") else
100           ALU_result_MEM; -- when (RegFile_Src_0 = "1x");
101
102 data_rd_MEM <= M4_out;
103
104 -- MUX S: define o dado a ser armazenado na memória em instr. SW, controlado
        pela unidade de Forward
105 MS_out <= fw_data_WB when (Forward_S = '1') else
106       data_rs2_MEM; -- when (Forward_S = '0');
107
108 -- Processo de escrita no registrador MEM/WB. Caso rst = '1' ou flush = '1',
        passa NOP
109 -- para o campo dos sinais de controle do registrador.
110 process (rst, clk)
111 begin
112   if (rising_edge(clk)) then
113     if (rst = '1') then
114       MEM_WB <= (others => '0');
115
116     -- prioridade maior sobre o sinal 'wait_data', e depois o 'flush_MEM_WB'
117     elsif (wait_data = '1') then
118         -- não escreve
119
120     elsif (wait_data = '0') then
121         ctrl_MEM_WB <= ctrl_EX_MEM & Mem_R; -- RegFile_Src_1 <= Mem_R
122         data_WB_0 <= M4_out;
123         data_WB_1  <= data_BUS;
124         funct3_WB <= funct3_MEM;
125         rd_WB   <= rd_MEM;
126
127     -- não utilizado no momento, entre passar NOP e reescrever novamente o
        mesmo resultado, se optou pelo último
```

```
128       --elsif (flush_MEM_WB = '1') then
129       -- ctrl_MEM_WB(1) <= '0'; -- NOP, bit correspondente ao sinal de controle
              RegFile_W
130
131       end if;
132
133     end if;
134   end process;
135
136 end architecture behavior;
```

F.17 – STAGE5_WB.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use work.core_pkg.all;
4
5  -- @file stage5_WB.vhd
6  -- @author Kevin Morais (moraisku@gmail)
7  -- @date 2018
8  -- @version 1
9  -- Write Back (WB), armazena o resultado no Banco de Registradores.
10
11 entity stage5_WB is
12   port(MEM_WB : in std_logic_vector(n_MEM_WB-1 downto 0); -- registrador de
         entrada do estágio Write Back
13     data_rd_WB : out std_logic_vector(n-1 downto 0)); -- dado a ser escrito no
          banco de regs.
14 end entity stage5_WB;
15
16 architecture behavior of stage5_WB is
17
18   -- identificação dos campos do registrador MEM/WB
19   alias RegFile_W : std_logic is MEM_WB(72);          -- esta conectado
         diretamente em 'core.vhd'
20   alias RegFile_Src_1 : std_logic is MEM_WB(71);  -- campo que possui o sinal de
          controle RegFile_Src_1
21   alias data_0 : std_logic_vector(n-1 downto 0) is MEM_WB(70 downto 39); --
         campo que possui o dado 0
22   alias LW : std_logic_vector(n-1 downto 0) is MEM_WB(38 downto 7); -- campo que
```

```
      possui o dado buscado em instruções LW
  alias funct3 : std_logic_vector(2 downto 0) is MEM_WB(6 downto 4);  -- campo
      funct3 da instrução, define a ação a ser tomada com o dado acessado da MEM
      (LW/LH/LB(U))
  alias rd_WB : std_logic_vector(3 downto 0)  is MEM_WB(3 downto 0);  -- esta
      conectado diretamente em 'core.vhd'

    signal L_ext : std_logic;      -- bit utilizado para extender o valor
       carregado da memória, sendo escolhido entre 0 ou o msb do dado acessado (
       ext de sinal)
    signal L_upper : std_logic_vector(31 downto 16); -- 16 bits superiores do
       sinal buscado da memória para instruções LH/LB(U)
    signal L_lower : std_logic_vector(15 downto 0); -- 16 bits inferiores do
       sinal buscado da memória para instruções LH/LB(U)

begin

  -- MUX 5 : escolhe a entrada de dados do banco de regs. entre data_0 e LW
  data_rd_WB <= data_0 when (RegFile_Src_1 = '0') else
                L_upper&L_lower; -- when (RegFile_Src_1 = '1');


    -- Lógica das instruções LW/LH/LB
    L_ext <= '0' when (funct3(2) = '1') else
             LW(15) when (funct3(0) = '1') else
             LW(7) when (funct3(0) = '0') else
       '0';

    L_upper <= LW(31 downto 16) when (funct3(1) = '1') else
               (others => L_ext);

    L_lower(15 downto 8) <= LW(15 downto 8) when (funct3(1) = '1' OR funct3(0) =
        '1') else
                        (others => L_ext); -- when (funct3(0) = '0');

    L_lower(7 downto 0) <= LW(7 downto 0);


end architecture behavior;
```

F.18 – TESTBENCH.VHD

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity testbench is
7    generic(n  : integer := 32; -- tamanho dos dados
8        n_addrIO: integer := 16); -- tamanho do end. de acesso aos disp. IO
9  end entity testbench;
10
11 architecture behavior of testbench is
12
13   -- Núcleo RV32EC
14   COMPONENT core
15     port(clk  : in std_logic;
16       rst   : in std_logic;
17       wait_data : in std_logic;
18       wait_instr  : in std_logic;
19       rvc   : in std_logic;
20       we   : out std_logic;
21       re   : out std_logic;
22       sb_en  : out std_logic;
23       sh_en  : out std_logic;
24       Valid_addr  : out std_logic;
25       addr_CPU : out std_logic_vector(n-1 downto 0);
26       data_CPU : inout std_logic_vector(n-1 downto 0));
27   end COMPONENT core;
28
29   -- Componente Decodificador da Memória
30   COMPONENT DEC is
31         port (rst : in std_logic;
32             clk : in std_logic;
33       we : in std_logic;
34       re   : in std_logic;
35             sb_en : in std_logic;
36             sh_en : in std_logic;
37             Valid_addr : in std_logic;
38             csIO : out std_logic;
39             wait_data : out std_logic;
```

```
40            wait_instr : out std_logic;
41            rvc : out std_logic;
42            addr_MEM : in std_logic_vector(n-1 downto 0);
43            data_MEM : inout std_logic_vector(n-1 downto 0));
44      end COMPONENT DEC;
45
46    -- Componente IO
47    COMPONENT IO is
48          port(rst : in std_logic;
49              clk : in std_logic;
50              csIO : in std_logic;
51              we_l : in std_logic;
52              addr : in std_logic_vector(n_addrIO-1 downto 0);
53              dataIO : inout std_logic_vector(n-1 downto 0);
54              RxD : in std_logic;
55              TxD : out std_logic);
56      end COMPONENT IO;
57
58  -- Sinais
59    signal clk : std_logic;       -- sinal de clock do sistema
60    signal rst : std_logic;       -- sinal de reset do sistema, ativo em '1'
61    signal wait_data : std_logic;       -- sinal de entrada da CPU que indica
          quando o dado da memória esta pronto
62    signal wait_instr : std_logic;       -- sinal de entrada da CPU que indica
          quando a instrução esta pronta
63    signal we : std_logic;       -- sinal write enable pela CPU, ativo em '1'
64    signal re  : std_logic;       -- read enable, sinal de habilitação de leitura
          de dados da memória, diferencia leitura de instr para dados na ROM
65    signal Valid_addr : std_logic;       -- sinal que indica quando o dado no
          barramento de endereço évalido, quando '1' = invalido
66    signal addr_BUS : std_logic_vector(n-1 downto 0); -- barramento de endereço do
           sistema, largura de 32 bits
67    signal data_BUS : std_logic_vector(n-1 downto 0); -- barramento de dados do
          sistema, largura de 32 bits
68
69    signal rvc  : std_logic;
70    signal sb_en : std_logic;
71    signal sh_en : std_logic;
72    signal csIO : std_logic;
73    signal RxD  : std_logic;
74    signal TxD  : std_logic;
```

```
75   signal we_l : std_logic;
76
77 begin
78
79     we_l <= NOT(we);
80   RxD <= '1';
81   TxD <= '1';
82
83   -- Processo de geração do sinal de clock de 100 MHz do tb.
84   signal_clk : process
85   begin
86
87     clk <= '0';  wait for 5 ns;
88     loop
89       clk <= '1'; wait for 5 ns;  -- Periodo = 10 ns , Freq = 100 MHz , Periodo/2
             = 5 ns
90       clk <= '0'; wait for 5 ns;
91     end loop;
92
93   end process signal_clk;
94
95   -- Processo de geração do sinal de reset
96   signal_rst : process
97   begin
98
99     rst <= '0';   wait for 16 ns;
100    loop
101      rst <= '1';  wait for 10 ns;
102      rst <= '0';  wait for 10000000 ns;
103      rst <= '0';  wait for 10 ms;
104
105    end loop;
106  end process signal_rst;
107
108 -- Port Map
109   -- instanciação do processador
110   core_i: core port map
111   (rst  => rst ,
112   clk  => clk ,
113   we   => we ,
114   re  => re ,
```

```
115   sb_en  => sb_en ,
116   sh_en => sh_en ,
117   wait_data => wait_data ,
118   wait_instr => wait_instr ,
119   rvc  => rvc ,
120   Valid_addr => Valid_addr ,
121   addr_CPU => addr_BUS ,
122   data_CPU => data_BUS);
123
124   -- instanciação da memória
125   DEC_i: DEC port map
126   (rst => rst ,
127   clk  => clk ,
128   we   => we ,
129   re  => re ,
130   sb_en  => sb_en ,
131   sh_en  => sh_en ,
132   Valid_addr => Valid_addr ,
133   csIO  => csIO ,
134   wait_data => wait_data ,
135   wait_instr => wait_instr ,
136   rvc  => rvc ,
137   addr_mem => addr_BUS ,
138   data_MEM => data_BUS);
139
140   -- instanciação do bloco IO
141     IO_i: IO port map
142     (rst  => rst ,
143     clk  => clk,
144     csIO  => csIO ,
145     addr  => addr_BUS(n_addrIO-1 downto 0) ,
146     dataIO  => data_BUS ,
147     we_l  => we_l ,
148   RxD  => RxD ,
149   TxD  => TxD);
150
151 end architecture behavior;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity top_module is
5    generic(n   : integer := 32; -- tamanho dos dados
6        n_addrIO: integer := 16); -- tamanho do end. de acesso aos disp. IO
7    port(clk : in std_logic; -- sinal de clock
8      rst_b : in std_logic; -- sinal de reset, passa por lógica de debouncing
9      led : out std_logic; -- sinal de saída conectado a LED da placa nexys3
10     RxD : in std_logic; -- sinal de recepção da UART
11     TxD : out std_logic); -- sinal de transmissão da UART
12 end entity top_module;
13
14 architecture behavior of top_module is
15
16   -- Núcleo Pipeline RV32EC
17   COMPONENT core
18     port(clk  : in std_logic;
19       rst   : in std_logic;
20       wait_data : in std_logic;
21       wait_instr  : in std_logic;
22       rvc   : in std_logic;
23       we   : out std_logic;
24       re    : out std_logic;
25       sb_en  : out std_logic;
26       sh_en  : out std_logic;
27       Valid_addr : out std_logic;
28       addr_CPU : out std_logic_vector(n-1 downto 0);
29       data_CPU : inout std_logic_vector(n-1 downto 0));
30   end COMPONENT core;
31
32   -- Componente Decodificador da Memória
33   COMPONENT DEC is
34         port (rst : in std_logic;
35             clk : in std_logic;
36       we : in std_logic;
37       re   : in std_logic;
38             sb_en : in std_logic;
39             sh_en : in std_logic;
40             Valid_addr : in std_logic;
41             csIO : out std_logic;
```

```
42          wait_data : out std_logic;
43          wait_instr : out std_logic;
44          rvc : out std_logic;
45          addr_MEM : in std_logic_vector(n-1 downto 0);
46          data_MEM : inout std_logic_vector(n-1 downto 0));
47    end COMPONENT DEC;
48
49  -- Componente IO
50  COMPONENT IO is
51       port(rst : in std_logic;
52          clk : in std_logic;
53          csIO : in std_logic;
54          we_l : in std_logic;
55          addr : in std_logic_vector(n_addrIO-1 downto 0);
56          dataIO : inout std_logic_vector(n-1 downto 0);
57          RxD : in std_logic;
58          TxD : out std_logic);
59    end COMPONENT IO;
60
61 -- Sinais
62  signal wait_data : std_logic;      -- sinal de entrada da CPU que indica
        quando o dado da memória esta pronto
63  signal wait_instr : std_logic;      -- sinal de entrada da CPU que indica
        quando uma instr. esta pronta
64  signal we : std_logic;      -- sinal write enable low provido pela CPU, ativo
        em '0'
65  signal re  : std_logic;      -- read enable, sinal de habilitação de leitura
        de dados da memória, diferencia leitura de instr para dados na ROM
66  signal Valid_addr : std_logic;      -- sinal que indica quando um dado no
        barramento de endereços évalido, quando = '1' invalido
67  signal addr_BUS : std_logic_vector(n-1 downto 0); -- barramento de endereço do
         sistema, largura de 32 bits
68  signal data_BUS : std_logic_vector(n-1 downto 0); -- barramento de dados do
        sistema, largura de 32 bits
69
70  signal rvc  : std_logic;
71  signal sb_en : std_logic;
72  signal sh_en : std_logic;
73  signal csIO : std_logic;
74  signal we_l : std_logic; -- uart funciona com write enable ativo em '0'
75
```

```
76   signal rst2led : std_logic;
77
78   -- Reset Button Debouncing
79   type State_Type is (S0, S1);
80   signal State : State_Type := S0;
81   signal rst : std_logic;
82   signal DPB, SPB : STD_LOGIC;
83   signal DReg : STD_LOGIC_VECTOR (7 downto 0);
84
85 begin
86
87   we_l <= NOT(we); -- usado com sinal invertido na seção IO (UART mais
         especificamente), ativo em '0'
88
89   rst2led <= rst;
90   led <= rst2led;
91
92   -- Reset Button Debouncing
93   process (clk , rst_b)
94   variable SDC : integer;
95   constant Delay : integer := 50000;
96     begin
97   if clk'Event and clk = '1' then
98     -- Double latch input signal
99     DPB <= SPB;
100    SPB <= rst_b;
101
102    case State is
103    when S0 =>
104      DReg <= DReg(6 downto 0) & DPB;
105
106      SDC := Delay;
107
108      State <= S1;
109    when S1 =>
110      SDC := SDC - 1;
111
112      if SDC = 0 then
113      State <= S0;
114      end if;
115    when others =>
```

```
116       State <= S0;
117     end case;
118
119     if DReg = X"FF" then
120     rst <= '1';
121     elsif DReg = X"00" then
122     rst <= '0';
123     end if;
124   end if;
125   end process;
126
127 -- Port Map
128   -- instanciação do processador
129   core_i: core port map
130   (rst  => rst ,
131   clk  => clk ,
132   we   => we ,
133   re   => re ,
134   sb_en  => sb_en ,
135   sh_en  => sh_en ,
136   wait_data => wait_data ,
137   wait_instr => wait_instr ,
138   rvc  => rvc ,
139   Valid_addr => Valid_addr ,
140   addr_CPU => addr_BUS ,
141   data_CPU => data_BUS);
142
143   -- instanciação do decodificador
144   DEC_i: DEC port map
145   (rst  => rst ,
146   clk  => clk ,
147   we   => we ,
148   re  => re ,
149   sb_en => sb_en ,
150   sh_en  => sh_en ,
151   Valid_addr => Valid_addr ,
152   csIO  => csIO ,
153   wait_data => wait_data ,
154   wait_instr => wait_instr ,
155   rvc  => rvc ,
156   addr_mem => addr_BUS ,
```

```
157   data_MEM => data_BUS);
158
159   -- instanciação do bloco IO
160     IO_i: IO port map
161     (rst  => rst ,
162     clk  => clk ,
163     csIO  => csIO ,
164     addr  => addr_BUS(n_addrIO-1 downto 0) ,
165     dataIO  => data_BUS ,
166     we_l  => we_l ,
167   RxD  => RxD ,
168   TxD  => TxD);
169
170 end architecture behavior;
```

F.20 – RV32EC_CORE.UCF

```
1  ##Clock signal
2  Net "clk" LOC=V10 |IOSTANDARD=LVCMOS33;
3  Net "clk" TNM_NET = sys_clk_pin;
4  TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;
5
6  # Reset Switch
7  Net "rst" LOC = T10 | IOSTANDARD = LVCMOS33; #Bank = 2, pin name = IO_L29N_GCLK2
       , Sch name = SW0
8
9  # Inputs and Outputs
10 INST "data_CPU<0>" TNM = input_pad_group;
11 INST "data_CPU<1>" TNM = input_pad_group;
12 INST "data_CPU<2>" TNM = input_pad_group;
13 INST "data_CPU<3>" TNM = input_pad_group;
14 INST "data_CPU<4>" TNM = input_pad_group;
15 INST "data_CPU<5>" TNM = input_pad_group;
16 INST "data_CPU<6>" TNM = input_pad_group;
17 INST "data_CPU<7>" TNM = input_pad_group;
18 INST "data_CPU<8>" TNM = input_pad_group;
19 INST "data_CPU<9>" TNM = input_pad_group;
20 INST "data_CPU<10>" TNM = input_pad_group;
21 INST "data_CPU<11>" TNM = input_pad_group;
22 INST "data_CPU<12>" TNM = input_pad_group;
```

```
INST "data_CPU<13>" TNM = input_pad_group;
INST "data_CPU<14>" TNM = input_pad_group;
INST "data_CPU<15>" TNM = input_pad_group;
INST "data_CPU<16>" TNM = input_pad_group;
INST "data_CPU<17>" TNM = input_pad_group;
INST "data_CPU<18>" TNM = input_pad_group;
INST "data_CPU<19>" TNM = input_pad_group;
INST "data_CPU<20>" TNM = input_pad_group;
INST "data_CPU<21>" TNM = input_pad_group;
INST "data_CPU<22>" TNM = input_pad_group;
INST "data_CPU<23>" TNM = input_pad_group;
INST "data_CPU<24>" TNM = input_pad_group;
INST "data_CPU<25>" TNM = input_pad_group;
INST "data_CPU<26>" TNM = input_pad_group;
INST "data_CPU<27>" TNM = input_pad_group;
INST "data_CPU<28>" TNM = input_pad_group;
INST "data_CPU<29>" TNM = input_pad_group;
INST "data_CPU<30>" TNM = input_pad_group;
INST "data_CPU<31>" TNM = input_pad_group;
INST "clk" TNM = input_pad_group;
INST "rst" TNM = input_pad_group;
INST "rvc" TNM = input_pad_group;
INST "wait_data" TNM = input_pad_group;
TIMEGRP "input_pad_group" OFFSET = IN 10 ns VALID 10 ns BEFORE "clk" RISING;
INST "data_CPU<0>" TNM = output_pad_group;
INST "data_CPU<1>" TNM = output_pad_group;
INST "data_CPU<2>" TNM = output_pad_group;
INST "data_CPU<3>" TNM = output_pad_group;
INST "data_CPU<4>" TNM = output_pad_group;
INST "data_CPU<5>" TNM = output_pad_group;
INST "data_CPU<6>" TNM = output_pad_group;
INST "data_CPU<7>" TNM = output_pad_group;
INST "data_CPU<8>" TNM = output_pad_group;
INST "data_CPU<9>" TNM = output_pad_group;
INST "data_CPU<10>" TNM = output_pad_group;
INST "data_CPU<11>" TNM = output_pad_group;
INST "data_CPU<12>" TNM = output_pad_group;
INST "data_CPU<13>" TNM = output_pad_group;
INST "data_CPU<14>" TNM = output_pad_group;
INST "data_CPU<15>" TNM = output_pad_group;
INST "data_CPU<16>" TNM = output_pad_group;
```

```
INST "data_CPU<17>" TNM = output_pad_group;
INST "data_CPU<18>" TNM = output_pad_group;
INST "data_CPU<19>" TNM = output_pad_group;
INST "data_CPU<20>" TNM = output_pad_group;
INST "data_CPU<21>" TNM = output_pad_group;
INST "data_CPU<22>" TNM = output_pad_group;
INST "data_CPU<23>" TNM = output_pad_group;
INST "data_CPU<24>" TNM = output_pad_group;
INST "data_CPU<25>" TNM = output_pad_group;
INST "data_CPU<26>" TNM = output_pad_group;
INST "data_CPU<27>" TNM = output_pad_group;
INST "data_CPU<28>" TNM = output_pad_group;
INST "data_CPU<29>" TNM = output_pad_group;
INST "data_CPU<30>" TNM = output_pad_group;
INST "data_CPU<31>" TNM = output_pad_group;
INST "addr_CPU<0>" TNM = output_pad_group;
INST "addr_CPU<1>" TNM = output_pad_group;
INST "addr_CPU<2>" TNM = output_pad_group;
INST "addr_CPU<3>" TNM = output_pad_group;
INST "addr_CPU<4>" TNM = output_pad_group;
INST "addr_CPU<5>" TNM = output_pad_group;
INST "addr_CPU<6>" TNM = output_pad_group;
INST "addr_CPU<7>" TNM = output_pad_group;
INST "addr_CPU<8>" TNM = output_pad_group;
INST "addr_CPU<9>" TNM = output_pad_group;
INST "addr_CPU<10>" TNM = output_pad_group;
INST "addr_CPU<11>" TNM = output_pad_group;
INST "addr_CPU<12>" TNM = output_pad_group;
INST "addr_CPU<13>" TNM = output_pad_group;
INST "addr_CPU<14>" TNM = output_pad_group;
INST "addr_CPU<15>" TNM = output_pad_group;
INST "addr_CPU<16>" TNM = output_pad_group;
INST "addr_CPU<17>" TNM = output_pad_group;
INST "addr_CPU<18>" TNM = output_pad_group;
INST "addr_CPU<19>" TNM = output_pad_group;
INST "addr_CPU<20>" TNM = output_pad_group;
INST "addr_CPU<21>" TNM = output_pad_group;
INST "addr_CPU<22>" TNM = output_pad_group;
INST "addr_CPU<23>" TNM = output_pad_group;
INST "addr_CPU<24>" TNM = output_pad_group;
INST "addr_CPU<25>" TNM = output_pad_group;
```

```
INST "addr_CPU<26>" TNM = output_pad_group;
INST "addr_CPU<27>" TNM = output_pad_group;
INST "addr_CPU<28>" TNM = output_pad_group;
INST "addr_CPU<29>" TNM = output_pad_group;
INST "addr_CPU<30>" TNM = output_pad_group;
INST "addr_CPU<31>" TNM = output_pad_group;
INST "we" TNM = output_pad_group;
INST "sb_en" TNM = output_pad_group;
INST "sh_en" TNM = output_pad_group;
TIMEGRP "output_pad_group" OFFSET = OUT 10 ns AFTER "clk";
```

F.21 – RV32EC_SIST.UCF

```
##Clock signal
Net "clk" LOC=V10 |IOSTANDARD=LVCMOS33;
Net "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;

## Usb-RS232 interface
Net "RxD" LOC = N17 |IOSTANDARD=LVCMOS33; #Bank = 1, pin name =
    IO_L48P_HDC_M1DQ8, Sch name = MCU-RX
Net "TxD" LOC = N18 |IOSTANDARD=LVCMOS33; #Bank = 1, pin name = IO_L48N_M1DQ9,
    Sch name = MCU-TX
# Reset Button
Net "rst_b" LOC = C9 |IOSTANDARD = LVCMOS33; #Bank = 0, pin name =
    IO_L34N_GCLK18, Sch name = BTND
Net "led" LOC = U16 |IOSTANDARD = LVCMOS33; #Bank = 2, pin name = IO_L2P_CMPCLK,
     Sch name = LD0
# Input Group - OFFSET IN
INST "rst" TNM = input_pad_group;
INST "RxD" TNM = input_pad_group;
TIMEGRP "input_pad_group" OFFSET = IN 10 ns VALID 10 ns BEFORE "clk" RISING;

# Output Group - OFFSET OUT
INST "TxD" TNM = output_pad_group;
TIMEGRP "output_pad_group" OFFSET = OUT 10 ns AFTER "clk";

# Paths ignored by time analyzer:
# ROM -> RAM ; RAM -> BHT
INST "DEC_i/RAM_i/Mram_ram1" TNM = bram_ram_group;
```

```
24 INST "DEC_i/RAM_i/Mram_ram2" TNM = bram_ram_group;
25 INST "DEC_i/RAM_i/Mram_ram3" TNM = bram_ram_group;
26 INST "DEC_i/RAM_i/Mram_ram4" TNM = bram_ram_group;
27 INST "DEC_i/RAM_i/Mram_ram5" TNM = bram_ram_group;
28 INST "DEC_i/RAM_i/Mram_ram6" TNM = bram_ram_group;
29 INST "DEC_i/RAM_i/Mram_ram7" TNM = bram_ram_group;
30 INST "DEC_i/RAM_i/Mram_ram8" TNM = bram_ram_group;
31 INST "DEC_i_ROM_i/Mram_rom1" TNM = bram_rom_group;
32 INST "DEC_i_ROM_i/Mram_rom2" TNM = bram_rom_group;
33 INST "DEC_i_ROM_i/Mram_rom3" TNM = bram_rom_group;
34 INST "DEC_i_ROM_i/Mram_rom4" TNM = bram_rom_group;
35 INST "DEC_i_ROM_i/Mram_rom5" TNM = bram_rom_group;
36 INST "DEC_i_ROM_i/Mram_rom6" TNM = bram_rom_group;
37 INST "DEC_i_ROM_i/Mram_rom7" TNM = bram_rom_group;
38 INST "DEC_i_ROM_i/Mram_rom8" TNM = bram_rom_group;
39 INST "core_i/stage_1/Predict_unit_i/BRAM_BHT_i/Mram_RAM" TNM = bram_bht_group;
40 INST "core_i/stage_1/Predict_unit_i/BRAM_2/Mram_RAM" TNM = bram_bht_group;
41
42 TIMESPEC TS_ig_rom_2_ram = FROM "bram_rom_group" TO "bram_ram_group" TIG;
43 TIMESPEC TS_ig_ram_to_bht = FROM "bram_ram_group" TO "bram_bht_group" TIG;
```