REVISTA BRASILEIRA DE MECATRÔNICA
FACULDADE SENAI DE TECNOLOGIA MECATRÔNICA

PORTABILIDADE E CUSTOMIZAÇÃO DE UM SOFTCORE RISC-V EM FPGA

PORTABILITY AND CUSTOMIZATION OF A RISC-V SOFTCORE IN FPGA

Diego Salviano Nagai[1], [i]
Leandro Poloni Dantas[2], [ii]
Marcones Cleber Brito da Silva[3], [iii]
Luis Carlos Canno[4], [iv]
Fernando Simplicio de Sousa[5], [v]

Data de submissão: (21/11/2022) Data de aprovação: (18/09/2023)

RESUMO

As FPGAs são dispositivos lógicos capazes de implementar qualquer tipo de circuito digital, esta característica torna este dispositivo uma excelente ferramenta para o estudo e desenvolvimento em diversas áreas da eletrônica e da computação. Entretanto há algumas características inerentes a estes dispositivos que podem inviabilizar a sua utilização. O custo dos kits de desenvolvimento para FPGAs é maior que aquele observado nos kits para microcontroladores, além disso a complexidade dos ambientes de desenvolvimento e a incompatibilidade entre si também desfavorecem o uso desta ferramenta. Para evitar o custo de aquisição de um novo kit e também poupar o tempo gasto no aprendizado de um novo ambiente de desenvolvimento a cada novo projeto, torna-se imprescindível adotar um processo de portabilidade. A portabilidade consiste em migrar um projeto desenvolvido em uma plataforma ou dispositivo para outra, mantendo as mesmas funcionalidades. Neste contexto, este trabalho teve como objetivo explorar a portabilidade de um *softcore* RISC-V em FPGA. Isso envolveu, primeiramente, a revisão de código e simulação do projeto a ser portado e, em seguida, a adequação do projeto à nova plataforma na qual o sistema foi embarcado. Utilizando o kit de desenvolvimento *DE10-Lite* e o ambiente *Intel Quartus Prime Lite Edition*, as funcionalidades originais do projeto foram mantidas após a portabilidade, e novos recursos foram adicionados. Dentre eles, chaves e displays de 7 segmentos como novos dispositivos de entrada/saída.

Palavras-chave: *softcore;* RISC-V; FPGA; portabilidade; customização.

[1] Pós-graduado na Faculdade de Tecnologia SENAI São Paulo – Campus “Anchieta”. E-mail: diego_nagai@terra.com.br
[2] Professor Dr. na Faculdade de Tecnologia SENAI São Paulo – Campus “Anchieta”. E-mail: leandro.poloni@sp.senai.br
[3] Professor Me. na Faculdade de Tecnologia SENAI São Paulo – Campus “Anchieta”. E-mail: marcones.silva@sp.senai.br
[4] Professor Esp. na Faculdade de Tecnologia SENAI São Paulo – Campus “Anchieta”. E-mail: luis.canno@sp.senai.br
[5] Professor Me. na Faculdade de Tecnologia SENAI São Paulo – Campus “Anchieta”. E-mail: fernando.simplicio@sp.senai.br

ABSTRACT

FPGAs are logic devices capable of implementing any type of digital circuit, this characteristic makes these devices an excellent tool for study and development in several areas of electronics and computing. However, there are some characteristics inherent to this device that may make its use unfeasible. The cost of FPGA development kits is higher than that observed for microcontroller kits, in addition to the complexity of development environments and their incompatibility with each other also disfavor the use of this tool. To avoid the cost of acquiring a new kit and also save time spent on learning a new development environment for each new project, it is essential to adopt a portability process. Portability involves migrating a project developed on one platform or device to another while maintaining the same functionalities. In this context, this study aimed to explore the portability of a RISC-V softcore on an FPGA. This encompassed, firstly, the code review and simulation of the project to be migrated, followed by adapting the project to the new platform where the system was embedded. Using the DE10-Lite development kit and the Intel Quartus Prime Lite Edition environment, the original project functionalities were maintained after porting, and new features were added. Among them, switches and 7-segment displays were incorporated as new input/output devices.

Keywords: softcore; RISC-V; FPGA; portability; customization.

1 INTRODUÇÃO

Um FPGA (*field programmable gate array*) é um dispositivo lógico programável, um tipo de circuito integrado que pode ser usado para implementar qualquer circuito digital. Estas características tornam este dispositivo uma excelente ferramenta de estudo e de desenvolvimento para diversas áreas da eletrônica e da computação.

Entretanto, há algumas características inerentes a este dispositivo que podem inviabilizar a sua utilização como uma ferramenta didática e de desenvolvimento. Dentre elas destacamos, o fato do custo dos kits de desenvolvimento para FPGAs ser maior que aquele observado nos kits para microcontroladores, a complexidade dos ambientes de desenvolvimento e a incompatibilidade entre os diferentes fabricantes. Assim, mesmo em posse de um kit de desenvolvimento com FPGA não é possível prototipar de maneira imediata projetos desenvolvidos em outros ambientes de desenvolvimento.

Para viabilizar a utilização de FPGAs, é necessário adotar uma metodologia que vise o processo de portabilidade. A portabilidade consiste em migrar um projeto desenvolvido em uma plataforma ou dispositivo para outra, mantendo as mesmas funcionalidades. Desta forma, é possível evitar o custo de aquisição de um novo kit de desenvolvimento e também poupar o tempo gasto no aprendizado de um novo ambiente de desenvolvimento a cada novo projeto.

O propósito deste estudo é demonstrar a portabilidade de um softcore RISC-V, inicialmente desenvolvido utilizando ferramentas de código aberto, da placa de desenvolvimento *iCEStick*, que incorpora um FPGA Lattice iCE40, para uma placa *DE10-Lite* com um FPGA Intel MAX10, fazendo uso de ferramentas oficiais da Intel. Além de assegurar a manutenção das funcionalidades originais do projeto, este trabalho também visa implementar periféricos customizados, especialmente ajustados para placa *DE10-Lite*.

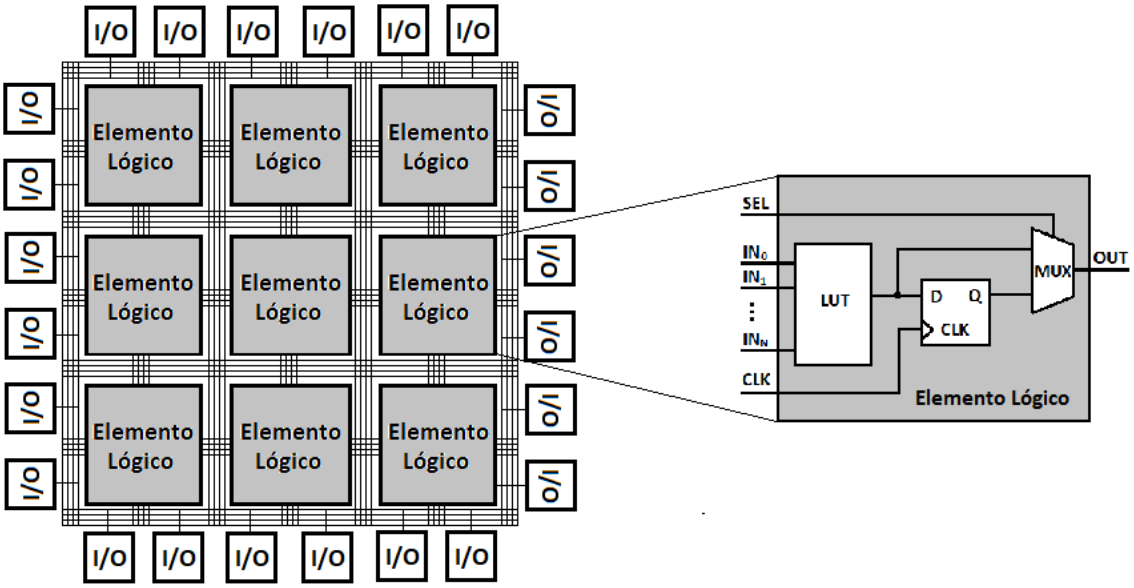
2 REVISÃO DA LITERATURA

Esta seção apresenta brevemente os principais conceitos que foram necessários para o desenvolvimento deste trabalho. Inicialmente, discute a estrutura típica das FPGAs e apresenta as linguagens de descrição de hardware (HDLs, *hardware description languages*) mais utilizadas. Esta seção também introduz o conceito de arquitetura de computadores e o papel fundamental da arquitetura do conjunto de instruções (ISA, *instruction set architecture*). Por fim, apresenta o conjunto base da arquitetura RISC-V e os *softcores*, processadores implementados em FPGA, que permitem uma grande flexibilidade de projeto.

2.1 FPGAs

Um FPGA é um dispositivo lógico programável que pode ser usado para implementar qualquer tipo de circuito digital, graças a sua estrutura interna. Segundo Ilda (2018), um FPGA consiste em três componentes básicos: o elemento lógico, o dispositivo de entrada e saída e a matriz de interconexões. A Figura 1 apresenta a estrutura típica de um FPGA.

Figura 1 – Estrutura típica de um FPGA.



Fonte: Elaborado pelo autor.

O elemento lógico pode representar qualquer função lógica utilizando basicamente três componentes: uma LUT (*look-up table*) para representar um circuito combinacional, um FF (*flip-flop*) para armazenar estados em circuitos sequenciais e um MUX (multiplexador) para seleção entre as saídas da LUT e do FF. O dispositivo de entrada e saída fornece uma conexão com o mundo exterior permitindo a comunicação do FPGA com outros dispositivos. Já a matriz de interconexões conecta os diferentes elementos da estrutura interna via roteamento programável (Floyd, 2009).

Além dos elementos básicos da estrutura, existem também diversos componentes opcionais especializados como blocos de processamento digital de sinal, utilizados para operações matemáticas, blocos de memória, utilizados para armazenamento massivo de dados e até mesmo processadores embarcados (Ilda, 2018).

2.2 Linguagens de descrição de hardware

As HDLs são utilizadas no projeto de circuitos eletrônicos, substituindo os diagramas esquemáticos por descrições textuais destes mesmos circuitos. Diferentemente das linguagens de programação que geralmente implementam um algoritmo, as HDLs implementam um hardware. Através delas, é possível descrever a estrutura de qualquer circuito digital e seu comportamento funcional. As HDLs mais utilizadas são o VHDL e o Verilog (Brock, 2019).

O VHDL foi desenvolvido na década de oitenta pelo Departamento de Defesa dos Estados Unidos para documentar o projeto de circuitos integrados e substituir a utilização de diagramas esquemáticos. O Verilog, também da década de oitenta, foi desenvolvido pela empresa Gateway Design Automation como uma linguagem proprietária para o projeto de circuitos integrados (Brock, 2019).

Ambas as linguagens são suportadas nos diferentes ambientes de desenvolvimento para FPGAs e são normatizadas pelo IEEE (*Institute of Electrical and Electronics Engineers*), sendo o padrão IEEE 1364-2001 para o Verilog e o IEEE 1076-2008 para o VHDL.

2.3 Arquitetura de Computadores

A arquitetura de computadores é um modelo abstrato que define a estrutura e as especificações de um sistema computacional. Ela é descrita por meio de uma ISA, um conjunto de instruções que atua como uma interface entre o software e o hardware. Existem diversos tipos de arquiteturas, como: RISC-V, ARM, x86, MIPS, SPARC e PowerPC, entretanto concentraremos o estudo deste trabalho na arquitetura RISC-V (Harris e Harris, 2021).

Embora a ISA descreva todas as especificações, tanto para o hardware como para o software, segundo Harris e Harris (2021) uma arquitetura de computador não define a implementação de hardware, por isso, podem existir diferentes implementações de hardware para uma mesma arquitetura. Por exemplo, Intel e AMD, ambos comercializam microprocessadores pertencentes a mesma arquitetura x86. Estes microprocessadores podem executar o mesmo software, mesmo possuindo diferentes hardwares em sua implementação.

2.4 *Softcores*

Diferentemente dos processadores fabricados em silício (*hardcores*), um *softcore* é descrito por meio de uma HDL e então é sintetizado e mapeado em um FPGA. Esta metodologia oferece uma maior flexibilidade pois um *softcore* pode ser customizado com a adição ou remoção de recursos de acordo com as necessidades da aplicação. Segundo Chu (2012), com desenvolvimento atual das FPGAs e a disponibilidade de processadores *softcore* é possível desenvolver e simular rapidamente hardware e software personalizados para construir sistemas embarcados sofisticados.

Os fabricantes de FPGAs disponibilizam seus próprios *softcores* em seu ambiente de desenvolvimento. Por exemplo, temos o MicroBlaze da Xilinx e o Nios II da Intel, que possuem arquiteturas proprietárias (Parab; Gad; Naik, 2018). No entanto, além dos *softcores* proprietários, existem também diversas opções código aberto, disponíveis em diferentes arquiteturas, incluindo a arquitetura RISC-V. Esta última tem se tornado a principal escolha no

desenvolvimento de *softcores*, devido à sua característica modular e à sua natureza livre de *royalties.*

2.5 RISC-V

O RISC-V é a quinta geração da arquitetura RISC (*reduced instruction set computer*) e foi desenvolvido em 2010 na universidade de Berkeley na California por Krste Asanović, Andrew Waterman e David Patterson. Suas principais características, como já mencionado, são a modularidade de uma ISA de código aberto e o modelo de licenciamento livre de *royalties*.

Segundo Patterson e Waterman (2017), no núcleo há uma ISA básica, chamada RV32I, que executa uma pilha completa de software. O RV32I está congelado e nunca será alterado, o que dá aos criadores de compiladores, desenvolvedores de sistemas operacionais e programadores de linguagem *assembly* um destino estável. A modularidade vem de extensões padrão opcionais que o hardware pode incluir ou não, dependendo das necessidades da aplicação. Essa modularidade permite implementações flexíveis e incrementais.

A ISA RV32I é composta por um conjunto mínimo de instruções destinadas a implementações de 32 bits. O prefixo RV32 indica que as instruções são projetadas para trabalhar com registradores de 32 bits e o sufixo I indica que essas instruções operam apenas valores inteiros.

As instruções do conjunto RV32I são divididas em diferentes categorias, incluindo carregamento e armazenamento de dados, operações aritméticas e lógicas, controle de fluxo e manipulação de registradores. Algumas das instruções mais comuns incluídas no RV32I são:

- Instruções de carga e armazenamento (*load* e *store*): Permitem a transferência de dados entre a memória e os registradores.
- Instruções aritméticas: incluem adição, subtração, multiplicação, divisão e operações de deslocamento.
- Instruções lógicas: incluem operações de AND, OR, XOR e NOT para manipulação de bits.
- Instruções de controle de fluxo: incluem instruções de salto condicional e incondicional, permitindo a execução condicional de trechos de código.
- Instruções de manipulação de registradores: permitem mover dados entre registradores e realizar operações de transferência de bits.

Na Tabela 1, é possível visualizar toda a ISA RV32I. As três primeiras colunas da tabela apresentam partes dos códigos binários que foram as instruções. A quarta coluna apresenta o formata da instrução em linguagem *assembly*. Já as colunas cinco e seis descrevem como as instruções funcionam. Na parte de baixo da tabela, é exemplificado como os campos que compõe as instruções são posicionados de acordo com a variação do tipo de instrução.

A instrução do tipo R (R-Type) é utilizada para operações entre registradores, o tipo I (I-Type) em operações com imediatos de 12 bits, e operações de load, o tipo S (S-Type) é utilizado para as operações de store, o tipo B (B-Type) para desvios condicionais, o tipo U (U-Type) é utilizado para operações com imediatos de 20 bits e o tipo J (J-Type) para saltos incondicionais.

A construção de cada tipo de instrução é realizada utilizando codificações específicas que caracterizam a ISA como código de operação (op), endereço do registrador de destino (rd), distinção de funções de 3 bits (funct3), distinção de funções de 7 bits (funct7), endereço dos registradores de origem (rs1 e rs2) e número imediato (imm).

Tabela 1 – Conjunto de Instruções de 32 bits.

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 – rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$				rd	op	U-Type
$imm_{20,10:1,11,19:12}$				rd	op	J-Type

- `imm`: signed immediate in $\mathbf{imm}_{11:0}$
- `uimm`: 5-bit unsigned immediate in $\mathbf{imm}_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $\mathbf{imm}_{31:12}$
- `Address`: memory address: **rs1** + SignExt($\mathbf{imm}_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$\mathbf{imm}_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$\mathbf{imm}_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits

Fonte: Harris e Harris (2021).

3 METODOLOGIA

Nesta seção, são expostos os materiais e métodos empregados no desenvolvimento deste estudo. Inicialmente, são enfatizadas as principais características do *softcore* selecionado para o processo de portabilidade. Além disso, é introduzida a placa de desenvolvimento utilizada na implementação, bem como o ambiente de desenvolvimento associado. Por fim, é delineada a metodologia aplicada nas etapas de simulação e síntese do *softcore*.

3.1 Softcore FemtoRV

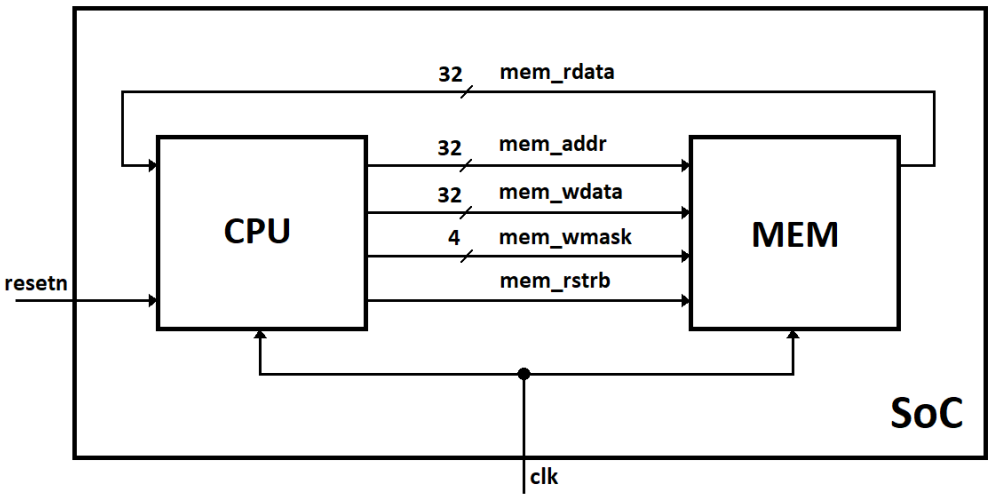
O *softcore* escolhido para o processo de portabilidade e customização foi o *FemtoRV*, desenvolvido por Bruno Levy, um dos embaixadores da comunidade RISC-V.

Bruno Levy é pesquisador do Inria, instituto francês de pesquisa em ciência digital e tecnologia. Ele é o diretor do centro de pesquisa Nancy Grand-Est. Como pesquisador, trabalha com física computacional, com aplicações em simulação de fluidos e cosmologia. Como entusiasta do RISC-V, ele é o principal autor e mantenedor do *FemtoRV*, um design RISC-V minimalista e fácil de entender sob medida para a educação (RISC-V INTERNATIONAL, 2022).

O *FemtoRV* é um design RISC-V minimalista, com código-fonte em Verilog fácil de ler e escrito diretamente da especificação RISC-V. A versão mais elementar (quark), um núcleo RV32I, tem apenas 400 linhas de código (versão documentada), e 100 linhas sem comentários. Existem também versões mais elaboradas, a maior (petitbateau) é um núcleo RV32IMFC. Também há um SoC complementar, com drivers para um UART, uma matriz de led, um pequeno display OLED, SPI RAM e SDCard (Levy, 2022).

O *FemtoRV* foi concebido como um SoC (*System on a Chip*), ou seja, um sistema completo em um único chip. Sua estrutura básica contempla uma unidade central de processamento (CPU – *central processing unit*) e uma unidade de memória integrados. A Figura 2 apresenta o diagrama de blocos do SoC.

Figura 2 – Diagrama de blocos do SoC *FemtoRV*.

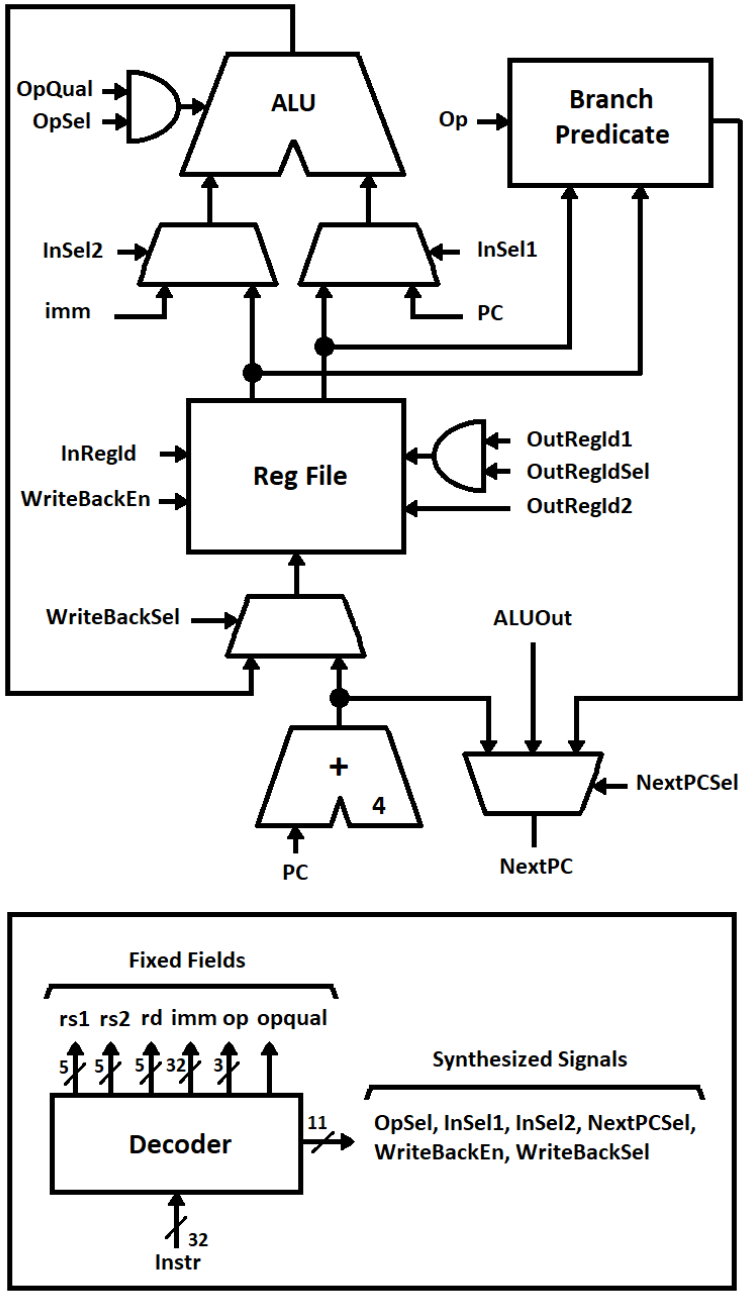


Fonte: Elaborado pelo autor.

A Figura 3 apresenta o diagrama em blocos da CPU com os elementos do *datapath*. Através dela, podemos observar detalhadamente a microarquitetura da CPU, onde é possível identificar os principais elementos do *datapath*: o *Decoder*, o *Reg File*, a ALU (*arithmetic logic unit*) e o *Branch Predicate*.

O *Decoder* é responsável pela decodificação das instruções. Ele também é responsável pela extensão dos imediatos e pela geração de sinais internos utilizados pela unidade de controle. O *Reg File* contém 32 registradores de 32 bits conforme a especificação RISC-V. A ALU é responsável pelas operações lógicas e aritméticas entre registradores, além realizar operações com imediatos e endereços de salto. O *Branch Predicate* é responsável pela sinalização do tipo de salto a ser executado, podendo ser condicional ou incondicional.

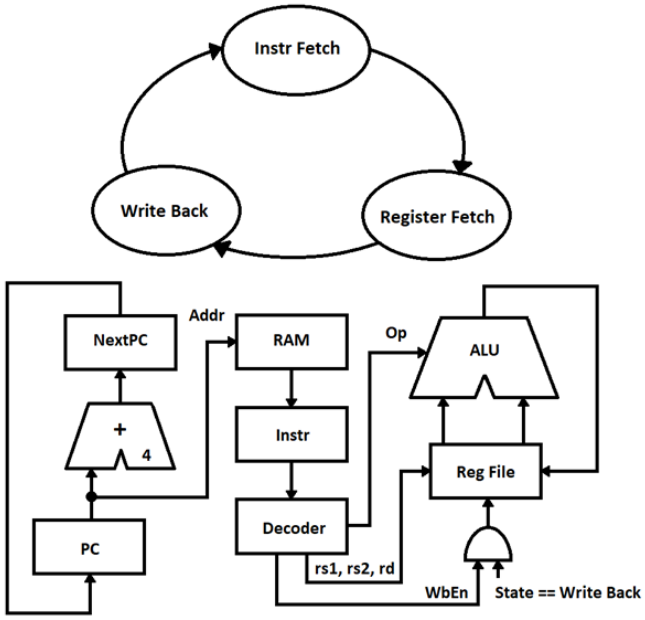
Figura 3 – Diagrama de blocos da CPU do *FemtoRV*.



Fonte: Levy (2022).

Os sinais que controlam o fluxo de dados (*datapath*) da Figura 3 são fornecidos pela unidade de controle. Esta unidade é composta por uma máquina de estados que executa a busca das instruções (*Instr Fetch*), endereça os registradores que serão operados (*Register Fetch*) e, por último, salva o resultado da operação entre os registradores (*Write Back*). O diagrama de estados dessa unidade de controle é apresentado na Figura 4.

Figura 4 – Diagrama de estados da unidade de controle.



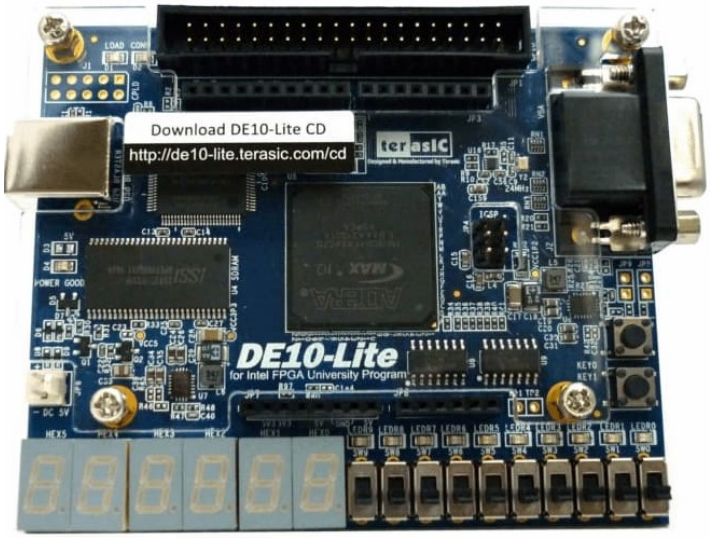
Fonte: Levy (2022).

No estado *Instr Fetch* é realizado o endereçamento da memória por meio do registrador *NextPC*, que armazena o endereço da próxima instrução a ser carregada no registrador *Instr*. No estado *Register Fetch*, a instrução armazenada no registrador *Instr* é então decodificada pelo *Decoder* e os registradores *rs1*, *rs2* e *rd* são endereçados no *Reg File*. No estado *Write Back*, o resultado processado pela ALU é, por fim, armazenado no registrador *rd* no *Reg File*. Este ciclo se repete para as demais instruções armazenadas na memória RAM.

3.2 Kit de desenvolvimento *DE10-Lite*

O hardware escolhido para a implementação da portabilidade foi a placa *DE10-Lite* da empresa Terasic, apresentado na Figura 5. Este kit possui um FPGA Altera/Intel MAX10 e conta com diversos periféricos que facilitaram a prototipagem (TERASIC, 2020).

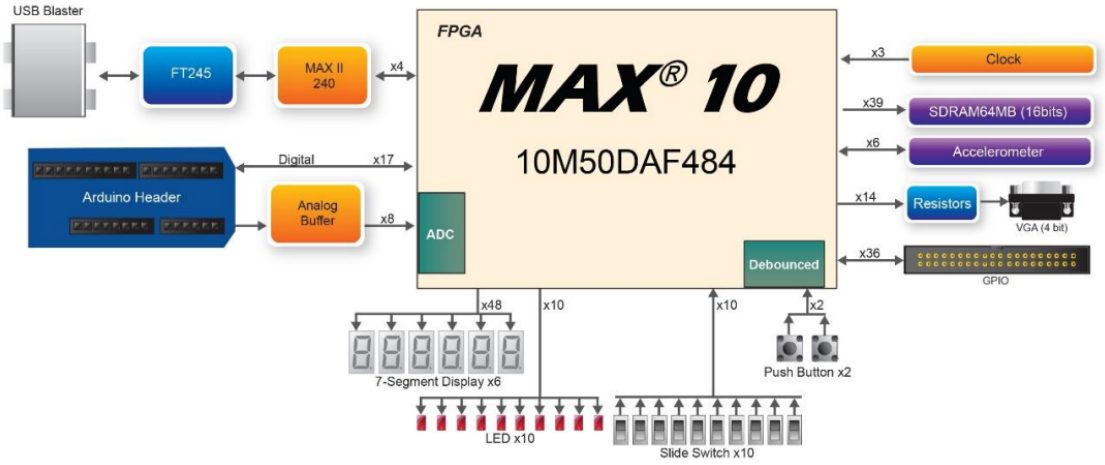
Figura 5 – Kit de desenvolvimento *DE10-Lite.*



Fonte: TERASIC (2020).

Na Figura 6, é apresentado o diagrama de blocos do kit mencionado. É possível observar a presença de um FPGA modelo 10M50DAF484, que possui cinquenta mil elementos lógicos. Além disso, o diagrama mostra uma variedade de periféricos externos ao FPGA, como uma memória SRAM de 64 MB, um conjunto de chaves e teclas, seis displays de sete segmentos e um acelerômetro.

Figura 6 – Diagrama de blocos do kit de desenvolvimento *DE10-Lite.*



Fonte: TERASIC (2020).

3.3 Software *Intel Quartus Prime*

O software *Intel Quartus Prime* é o ambiente oficial de desenvolvimento para FPGAs Altera/Intel. A versão escolhida para o trabalho foi a 18.1 *Lite Edition* (gratuita) (INTEL, 2018). O *Quartus Prime* é capaz de realizar a análise e a síntese de *designs* HDL, o que permite ao desenvolvedor compilar seus *designs*, realizar análises de tempo, examinar diagramas RTL (*register-transfer level*), simular uma resposta do design a diferentes estímulos e, por fim, gravar o FPGA no kit de desenvolvimento. O *Quartus Prime* permite a implementação por descrição de hardware em linguagem VHDL e Verilog, além de edição no nível de esquemático dos circuitos lógicos.

Com a versão 18.1, é possível implementar e utilizar todos os recursos disponíveis nos kits FPGA baseados nas famílias Arria II, Cyclone 10 LP, Cyclone IV, Cyclone V, MAX II, MAX V e MAX 10 FPGA (Nornberg, 2020).

3.4 Software *ModelSim*

O *ModelSim* é um software simulador de HDL desenvolvido pela Mentor Graphics. O pacote de instalação do *Intel Quartus Prime Lite Edition* 18.1 conta com uma versão também gratuita do *ModelSim*, a versão *Intel FPGA Starter Edition* 10.5, que foi a versão utilizada neste trabalho. Ele suporta a simulação das linguagens VHDL e Verilog e pode simular o código a nível de RTL e gate level. Em nível de RTL, o circuito é analisado em nível comportamental dos registradores, e em *gate level*, o circuito é analisado em seu nível mais baixo, conexão de portas lógicas (*netlist*), com a inclusão de tempos de atraso de propagação de sinais nas portas lógicas (Prado, 2014).

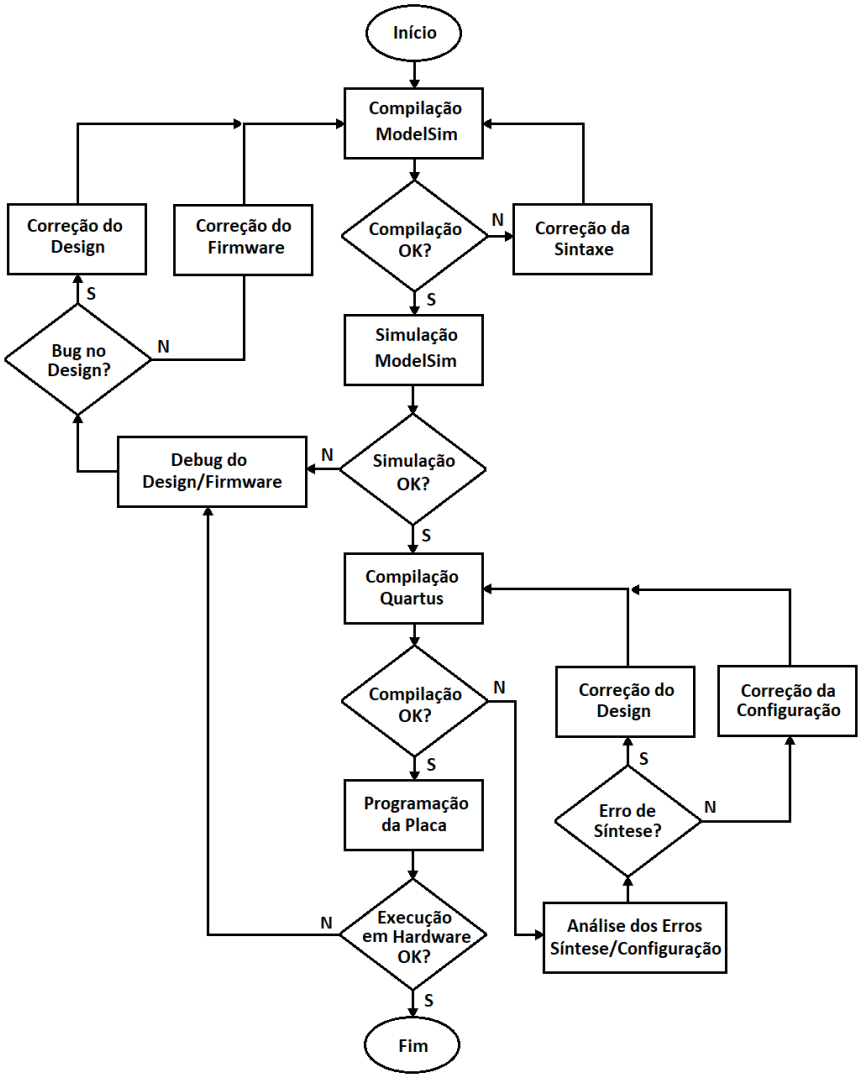
As simulações podem ser feitas utilizando apenas texto, com os estímulos e respostas em diferentes arquivos de texto ou graficamente, utilizando formas de onda para as análises

dos circuitos. Neste trabalho, foram utilizadas as formas de onda como ferramentas gráficas para análise do *softcore*.

3.5 Simulação

Inicialmente, o código HDL é compilado para verificar se não há erros de sintaxe. Caso existam erros, é necessário corrigi-los de acordo com a sintaxe Verilog. Com a compilação do código realizada com sucesso, é possível seguir com a fase de simulação. Na simulação, o funcionamento do *softcore* é avaliado graficamente, por meio das formas de onda dos sinais internos (GRAPHICS CORPORATION, 1991). Tanto o funcionamento do *core* como do *firmware* são avaliados nesta etapa. Caso algum erro de funcionamento seja encontrado, é iniciada a fase de depuração, onde, em primeiro lugar, verifica-se se há um problema no *design*, ou seja, no próprio processador e caso nenhum problema seja detectado segue-se com a verificação do *firmware*. Uma vez corrigidas as falhas, inicia-se novamente com a compilação e simulação do código. Este processo é repetido até o sucesso da simulação. Com a simulação bem-sucedida segue-se para a fase de síntese, que é realizada no ambiente do *Intel Quartus Prime*. O fluxo de simulação e síntese utilizado no projeto é apresentado na Figura 7.

Figura 7 – Fluxo de simulação e síntese.



Fonte: Elaborado pelo autor.

3.6 Síntese

Com o *softcore* devidamente simulado no *ModelSim*, inicia-se a compilação do projeto no *Intel Quartus Prime* para a realização da síntese e implementação. Esta nova compilação faz inicialmente uma verificação da sintaxe Verilog para a fase de síntese, que gera o circuito RTL, seguido pelo *place and route*, onde o circuito RTL é mapeado nos recursos internos do FPGA escolhido. Após o *place and route*, acontece a fase de *assembler*, que trabalha na geração do arquivo de programação do FPGA e, por fim, é realizada uma análise de tempo (*timing analysis*), onde, após o mapeamento no FPGA, os tempos de propagação do circuito são calculados (Bittencourt, 2021).

Conforme a Figura 7, caso algum erro de compilação ocorra, faz-se necessário verificar se o problema é relacionado a síntese ou a alguma configuração do FPGA. No caso de erro de síntese, é necessário avaliar se o código possui, por exemplo, estruturas não sintetizáveis e corrigi-las. Já para o caso de configurações do FPGA, os erros são avaliados um a um e corrigidos. O processo de correção pode ser feito com a utilização da página de ajuda do próprio ambiente *Intel Quartus Prime*. Uma vez realizada a compilação com sucesso, prossegue-se com a programação da placa e testes de funcionamento em hardware. Caso o funcionamento em hardware esteja correto, o processo é finalizado. Entretanto, caso o funcionamento em hardware não esteja de acordo com a simulação, o processo retorna para a fase de depuração (*debug*) do *design/firmware*. O processo se repete até que a implementação em hardware seja finalizada com sucesso.

4 RESULTADOS E DISCUSSÕES

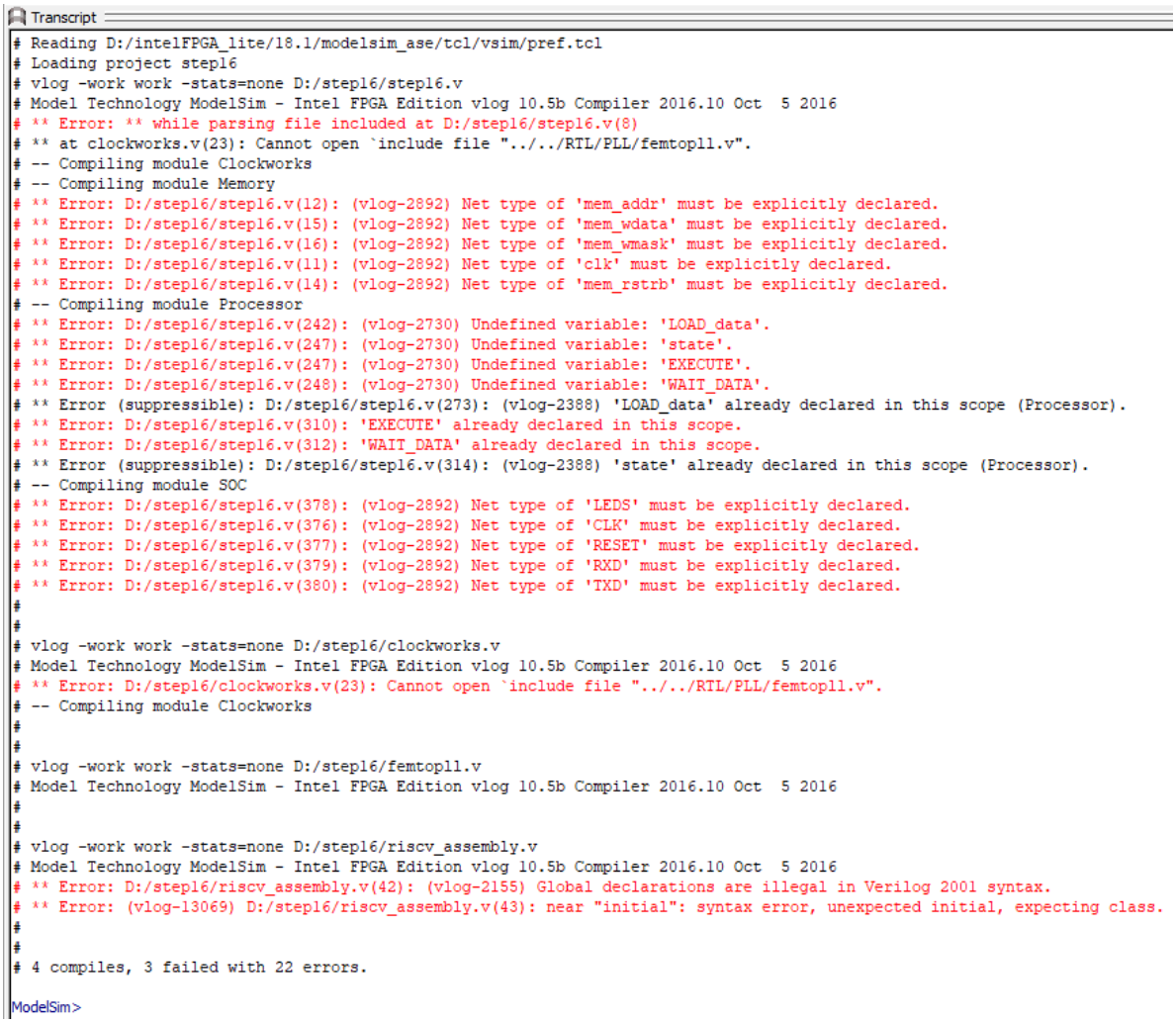
Nesta seção, são apresentados os resultados obtidos do processo de portabilidade e customização do *FemtoRV* para a placa *DE10-Lite* e são discutidas as principais dificuldades encontradas durante o desenvolvimento deste trabalho.

4.1 Portabilidade

Para poder entender as diferenças entre as plataformas, foi necessário iniciar com a simulação direta do projeto original. Foi adotado como ponto de partida o *step16.v* do tutorial *FROM_BLINKER_TO_RISCV*, disponível no *GitHub* do Bruno Levy[6]. Como esperado, foram observados vários erros de compilação no ambiente *ModelSim*, devido às discrepâncias entre os ambientes de desenvolvimento. A Figura 8 ilustra alguns dos erros encontrados durante a compilação do projeto.

[6] Repositório: https://github.com/BrunoLevy/learn-fpga/tree/master/FemtoRV/TUTORIALS/FROM_BLINKER_TO_RISCV

Figura 8 – Erros de compilação no *ModelSim*.



Fonte: Captura de tela do *ModelSim*.

Para corrigir os erros, apresentados na Figura 8, inicialmente foram removidos o módulo *clockworks.v* e o módulo *femtopll.v* do projeto. Estes módulos eram responsáveis pela criação do sinal de *clock* e faziam uso do PLL (*phase-locked loop*) interno do FPGA Lattice, incompatíveis com o FPGA MAX10. Nas simulações via *ModelSim*, foi utilizado um sinal de *clock* de 50 MHz criado dentro do módulo *soc_tb.v*.

Outro problema encontrado entre os ambientes de desenvolvimento foi o uso das *tasks* presentes no módulo *riscv_assembly.v*. Este módulo executava a montagem do código de máquina para a inicialização da memória do processador. Além disso, foi necessário explicitar os sinais apresentados pelo *report* declarando cada um como *reg* ou wire, de acordo com o seu contexto. Além destas modificações, foi necessário alterar a posição das variáveis de estados da máquina de estados para o início do código, devido às características do ambiente de simulação *ModelSim*.

A Figura 9 apresenta o uso das *tasks* na inicialização do módulo de memória.

Figura 9 – Inicialização da memória com o uso de *tasks.*

```
`include "riscv_assembly.v"
    integer L0_   = 12;
    integer L1_   = 40;
    integer wait_ = 64;
    integer L2_   = 72;

    initial begin

        LI(a0,0);
    // Copy 16 bytes from adress 400
    // to address 800
        LI(s1,16);
        LI(s0,0);
    Label(L0_);
        LB(a1,s0,400);
        SB(a1,s0,800);
        CALL(LabelRef(wait_));
        ADDI(s0,s0,1);
        BNE(s0,s1, LabelRef(L0_));
```

Fonte: Trecho de código do módulo *step16.v.*

Dentro do ambiente *ModelSim* as *tasks* não funcionaram e a memória era iniciada completamente zerada, ou seja, sem um programa a ser executado.

Para poder gerar o código de máquina para a inicialização da memória foi necessário utilizar o simulador online *Venus*, um simulador RISC-V desenvolvido por alunos da universidade de Berkley na Califórnia (Vakil, 2022).

O simulador *Venus* possibilita o desenvolvimento de programas em linguagem *assembly* RISC-V em diferentes extensões por meio de seu editor integrado e é capaz de realizar a simulação do código passo-a-passo ou de forma contínua no ambiente de simulação. Também é possível visualizar os dados contidos nos 32 registradores internos, bem como todos os dados contidos na memória.

Dentro do ambiente de simulação é possível visualizar o código fonte de entrada, o código básico gerado na extensão RV32I, o código de máquina no formato hexadecimal e o valor do contador de programa. A Figura 10 apresenta o ambiente do simulador *Venus*.

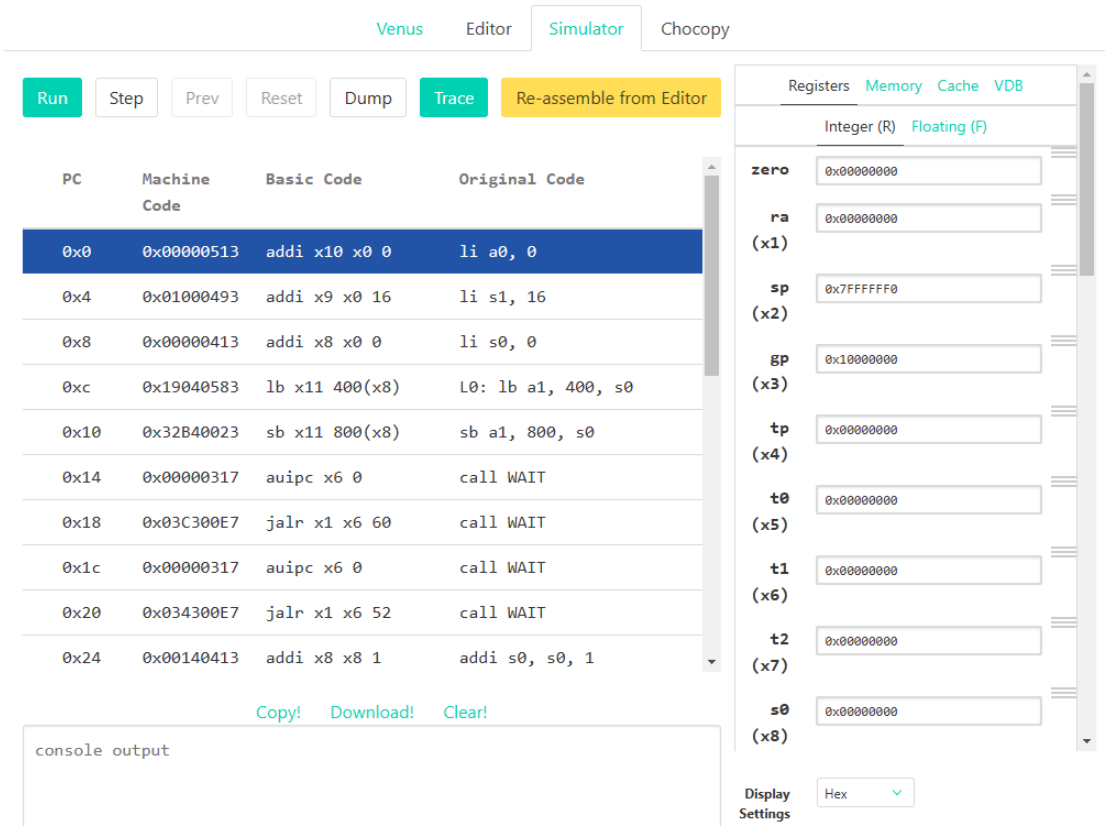
Todos os programas utilizados neste trabalho foram compilados com a utilização deste simulador. A inicialização do módulo de memória com o código de máquina gerado pelo *Venus Simulator* pode ser vista na Figura 11.

Uma vez resolvidos os problemas de compilação e inicialização da memória, foi possível executar a simulação do projeto original com sucesso.

O programa executado nesta simulação copia byte a byte o conteúdo dos endereços 100, 101, 102 e 103 para os endereços 200, 201, 202 e 203 da memória, usando as instruções *load byte* e *store byte*. Ao fim da cópia, o conteúdo dos endereços 200, 201, 202 e 203 são apresentados nos leds em binário.

O resultado da simulação é apresentado através do módulo *Waveforms* do *ModelSim* (Figura 12). Na figura é possível notar que no início da simulação os endereços de memória 200, 201, 202 e 203 não possuem nenhum valor válido armazenado, o que é representado por uma sequência de xs envoltos em linhas vermelhas no diagrama de tempos. Após alguns ciclos de *clock*, os valores presentes nos endereços 100, 101, 102 e 103 substituem os xs, à medida que o programa de inicialização da memória, representado na Figura 11, é executado. Isso confirma o correto funcionamento do *softcore*.

Figura 10 – Simulador *Venus*.



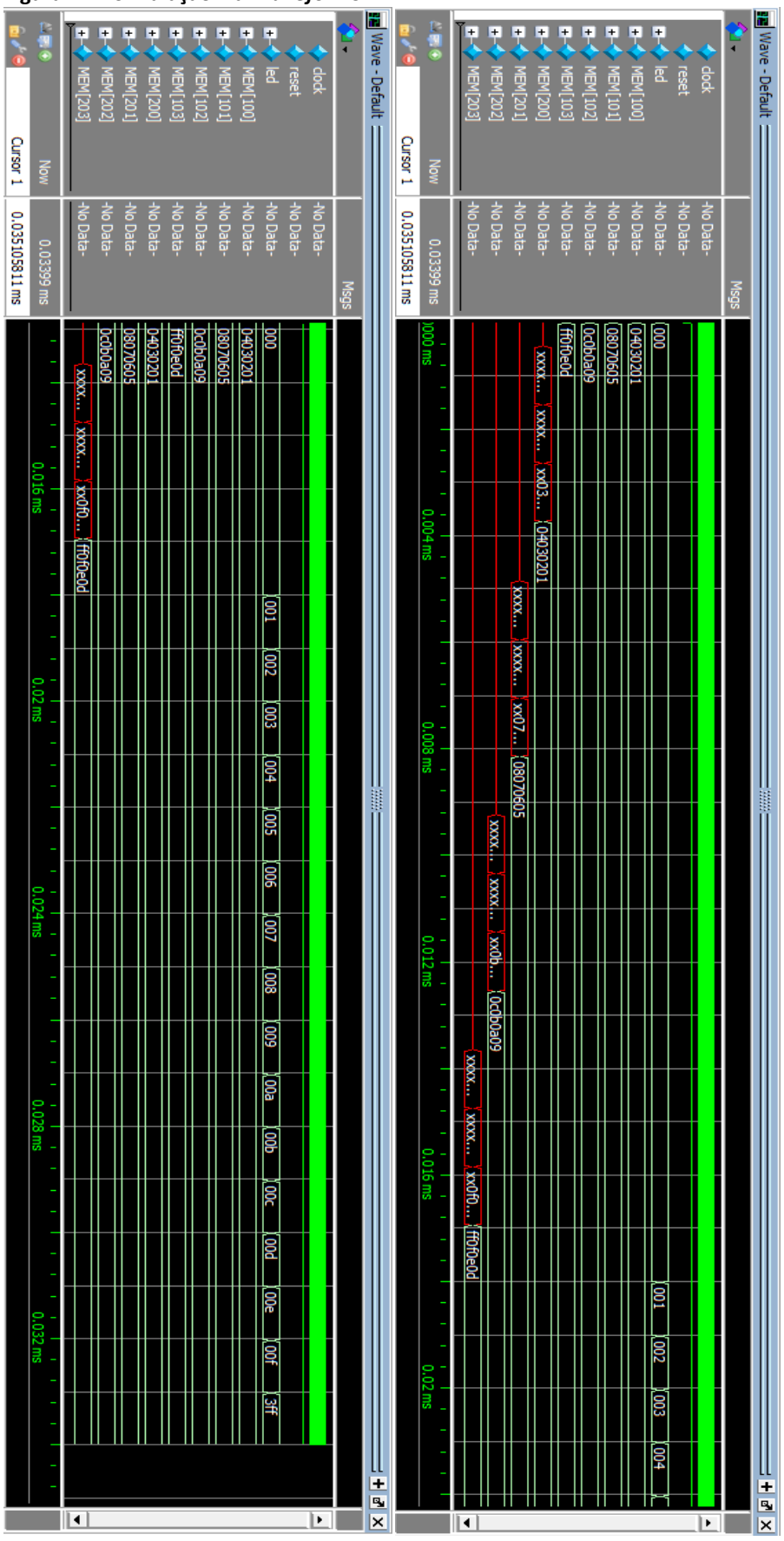
Fonte: Vakil (2022).

Figura 11 – Inicialização da memória com o código de máquina gerado no simulador *Venus*.

```
initial begin
 MEM[0]  = 32'h00000513; // addi x10 x0 0        li a0, 0
 MEM[1]  = 32'h01000493; // addi x9 x0 16        li s1, 16
 MEM[2]  = 32'h00000413; // addi x8 x0 0         li s0, 0
 MEM[3]  = 32'h19040583; // lb x11 400(x8)   L0: lb a1, 400, s0
 MEM[4]  = 32'h32B40023; // sb x11 800(x8)       sb a1, 800, s0
 MEM[5]  = 32'h00000317; // auipc x6 0           call WAIT
 MEM[6]  = 32'h02C300E7; // jalr x1 x6 44        call WAIT
 MEM[7]  = 32'h00140413; // addi x8 x8 1         addi s0, s0, 1
 MEM[8]  = 32'hFE9416E3; // bne x8 x9 -20        bne s0, s1, L0
 MEM[9]  = 32'h00000413; // addi x8 x0 0         li s0, 0
 MEM[10] = 32'h32040503; // lb x10 800(x8)   L1: lb a0, 800, s0
 MEM[11] = 32'h00000317; // auipc x6 0           call WAIT
 MEM[12] = 32'h014300E7; // jalr x1 x6 20        call WAIT
 MEM[13] = 32'h00140413; // addi x8 x8 1         addi s0, s0, 1
 MEM[14] = 32'hFE9418E3; // bne x8 x9 -16        bne s0, s1, L1
 MEM[15] = 32'h00100073; // ebreak               ebreak
 MEM[16] = 32'h00100293; // addi x5 x0 1   WAIT: li t0, 1
 MEM[17] = 32'h00129293; // slli x5 x5 1         slli t0, t0, 1
 MEM[18] = 32'hFFF28293; // addi x5 x5 -1        L2: addi t0, t0, -1
 MEM[19] = 32'hFE029EE3; // bne x5 x0 -4         bnez t0, L2
 MEM[20] = 32'h00008067; // jalr x0 x1 0         ret
 // Note: index 100 (word address)
 //     corresponds to
 // address 400 (byte address)
 MEM[100] = {8'h4, 8'h3, 8'h2, 8'h1};
 MEM[101] = {8'h8, 8'h7, 8'h6, 8'h5};
 MEM[102] = {8'hc, 8'hb, 8'ha, 8'h9};
 MEM[103] = {8'hff, 8'hf, 8'he, 8'hd};
end
```

Fonte: Trecho de código do módulo *memory.v*.

Figura 12 – Simulação via *Waveforms*.

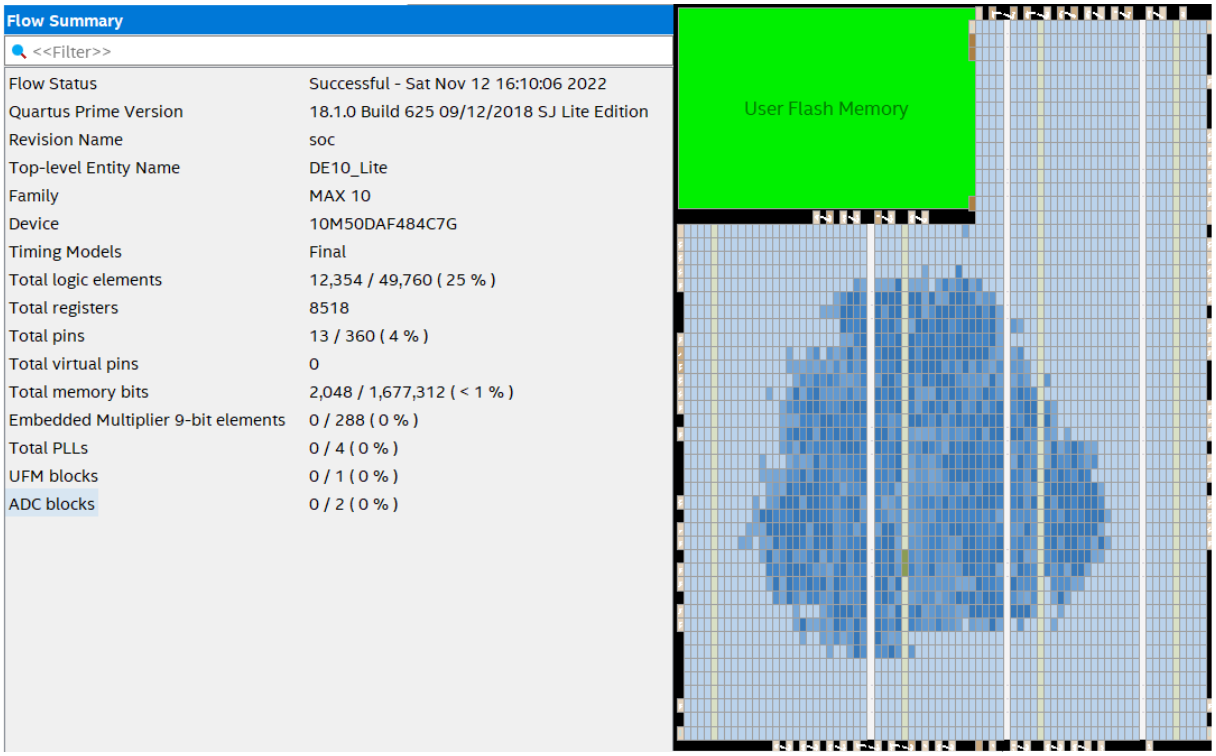


Fonte: Captura de tela do *ModelSim* (*Waveforms*).

Com a simulação bem-sucedida, o próximo passo foi a síntese, onde novamente mais um problema precisou ser resolvido. A implementação da memória realizada pela síntese do *Quartus* aconteceu completamente em *flip-flops*, diferentemente do Yosys que automaticamente infere o uso de BRAM (*Block RAM*) do FPGA Lattice.

Devido ao espaçamento em os blocos lógicos, a execução em hardware não funcionou como esperado, ocorrendo problemas com o tempo de propagação dos sinais (*timing*). Além disso, a implementação em *flip-flops* utilizou 25% dos elementos lógicos disponíveis no FPGA MAX10. A Figura 13 apresenta o relatório da síntese e o *place and route* do circuito sintetizado via *Chip Planner Viewer*. Nela, os retângulos em tons de azul mais escuros representam os elementos lógicos utilizados.

Figura 13 – Implementação da memória em *flip-flops*.



Fonte: *Intel Quartus Prime*.

Para resolver o problema de implementação da memória, foi utilizado o IP RAM-1PORT da Intel, que faz uso dos blocos internos de RAM no FPGA (INTEL, 2008). A Figura 14 apresenta a interface gráfica de configuração do IP (*Intellectual Property*), na qual é possível determinar as características como largura da palavra e profundidade da memória, entre outros.

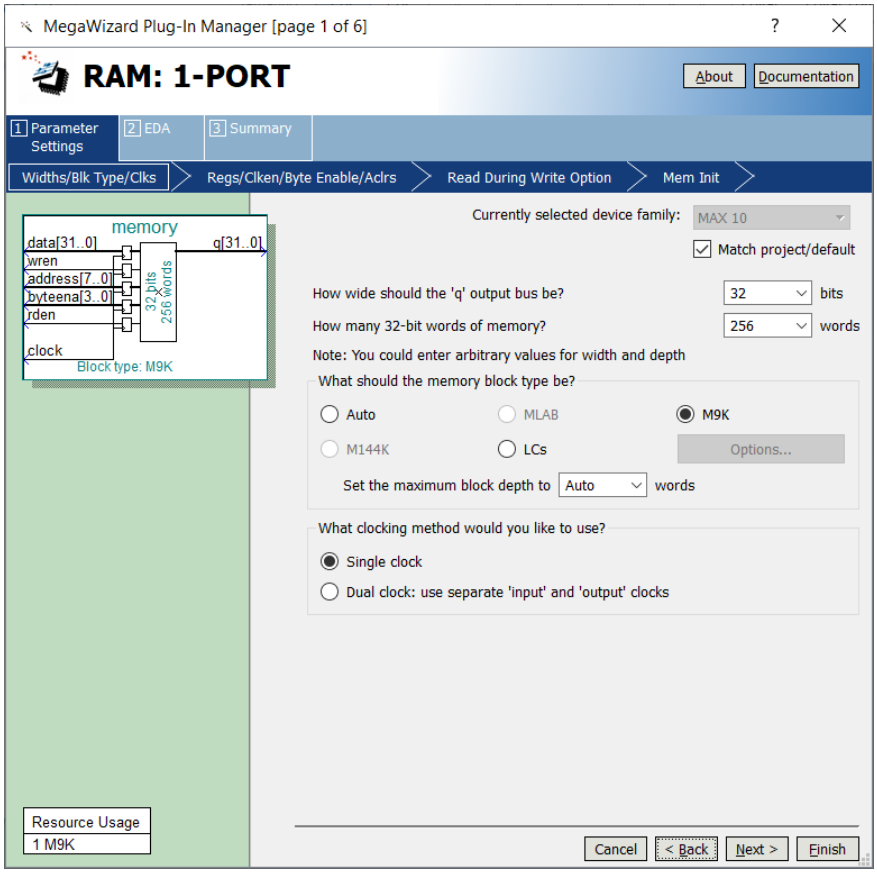
Os IPs são módulos funcionais pré-desenvolvidos que implementam funções específicas e otimizadas. A integração do IP RAM-1PORT com o projeto foi realizada diretamente via instanciação no código em Verilog, conforme a Figura 15.

O barramento de endereços (*.address*) recebeu apenas 8 bits do sinal *mem_addr*, pois a memória possuía 256 endereços no total. Além disso, os dois bits menos significativos do sinal *mem_addr* foram descartados, visto que a memória utilizada tem uma palavra de 32 bits, o que resulta em um endereçamento desalinhado de memória.

Destaca-se que, conforme especificação da ISA RISC-V, o contador de programa (PC) recebe um incremento de 4 bytes para acessar a instrução subsequente. Outro ponto a ser ressaltado está relacionado aos sinais de habilitação (*enable*) do IP. Dado que o IP utilizado

possui os sinais *read enable* (*.rden*) e *write enable* (*.wren*), optou-se por empregar o sinal *mem_rstrb* como controle para ambos os *enables*. Quando *mem_rstrb* está em nível alto, a memória entra em modo de leitura; enquanto que quando *mem_rstrb* está em nível baixo, a memória entra em modo de escrita. Os demais sinais foram diretamente conectados ao IP.

Figura 14 – Interface gráfica de configuração do IP de RAM.



Fonte: *Intel Quartus Prime*.

Figura 15 – Instanciação do IP RAM 1-PORT.

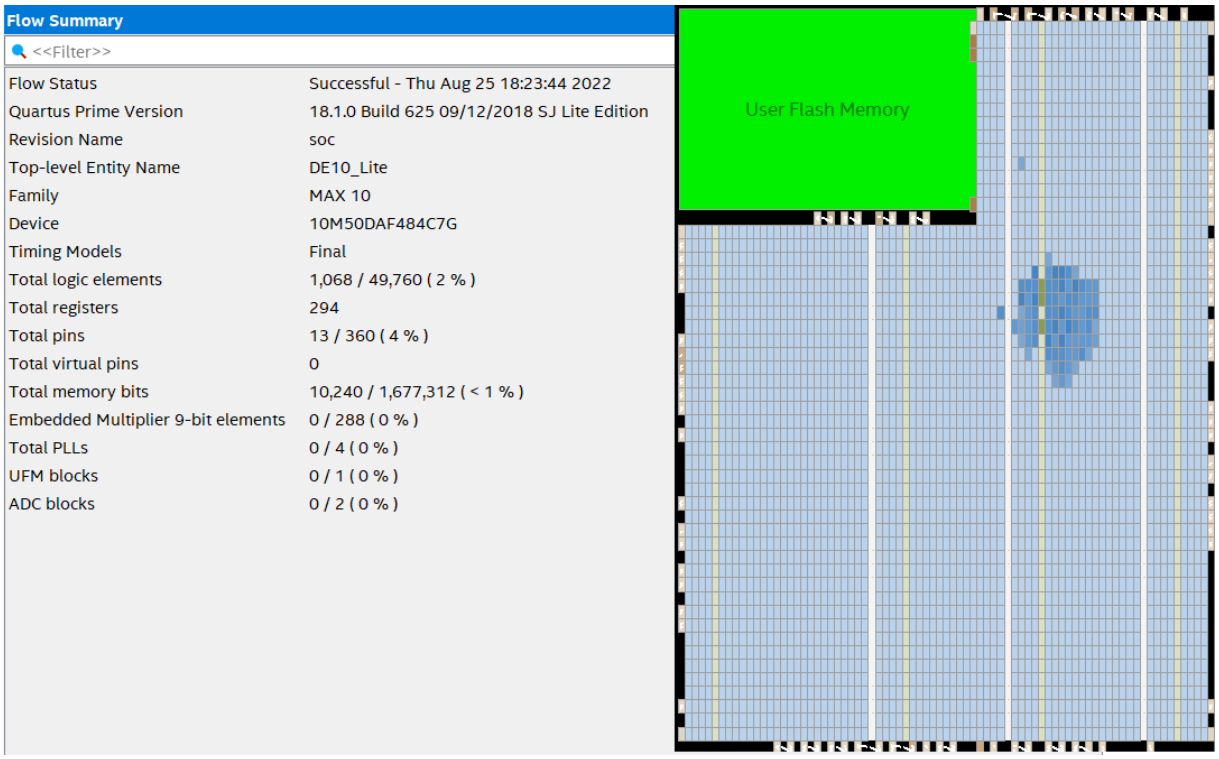
```
memory RAM(.address(w_address),
           .byteena(mem_wmask),
           .clock(clock),
           .data(mem_wdata),
           .rden(mem_rstrb),
           .wren(!mem_rstrb),
           .q(mem_rdata));

processor CPU(.clk(clock),
              .resetn(reset),
              .mem_addr(mem_addr),
              .mem_rdata(mem_rdata),
              .mem_rstrb(mem_rstrb),
              .mem_wdata(mem_wdata),
              .mem_wmask(mem_wmask),
              .x10(x10));
```

Fonte: Trecho de código do módulo *soc.v*.

Com a integração do IP RAM 1-PORT, o circuito sintetizado foi reduzido significativamente, utilizando apenas 2% dos elementos lógicos e menos de 1% dos blocos de memória RAM, conforme apresentado pela Figura 16. O uso de BRAM resolveu o problema de *timing* e a execução em hardware funcionou conforme a simulação.

Figura 16 – Implementação da memória em BRAM.



Fonte: *Intel Quartus Prime*.

4.2 Customização

Finalizada a portabilidade, iniciou-se a customização dos periféricos do *FemtoRV* para a placa *DE10-Lite*. Como referência, foi utilizado o arquivo *step17.v* do tutorial *FROM_BLINKER_TO_RISCV*, disponível no *GitHub* do Bruno Levy. No *step17*, o conceito de I/O mapeado em memória é abordado e o projeto passa por algumas mudanças na implementação.

O conceito de mapeamento em memória é baseado em segmentação das regiões de memória, onde cada região possui um uso específico. A Figura 17 mostra um exemplo de mapeamento comumente utilizado na arquitetura RISC-V.

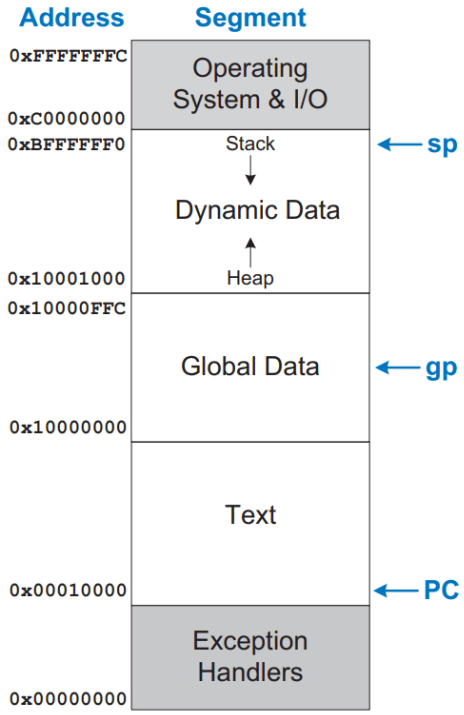
Papel de cada segmento:

- *Text*: armazena o programa em linguagem de máquina e é acessado pelo registrador PC (*program counter*);
- *Global Data*: armazena as variáveis globais e é acessado pelo registrador GP (*global pointer*);
- *Dynamic Data*: armazena variáveis locais na *Stack* e o *Heap* armazena dados alocados dinamicamente durante a execução do programa e são acessados pelo registrador SP (*stack pointer*);
- *Exceptions Handlers*: é utilizada para o *boot* de inicialização e também para o tratamento de interrupções, também é acessado pelo GP;

- *Operating System & I/O*: é reservada para chamadas de sistema no uso de sistemas operacionais e I/Os e também são acessados via GP (Harris e Harris, 2021).

Um ponto importante, a se destacar, é que os segmentos de memória mencionadas podem possuir diferentes tamanhos de acordo com a especificação dos sistemas e de acordo com a memória física disponível. Todas as regiões são alocadas fisicamente em memória exceto a região *Operating System & I/O*. Esta região pode ser tratada como uma região virtual, pois ela não existe fisicamente em memória, mas pode ser acessada pelo barramento de endereços via GP.

Figura 17 – Mapeamento de memória.



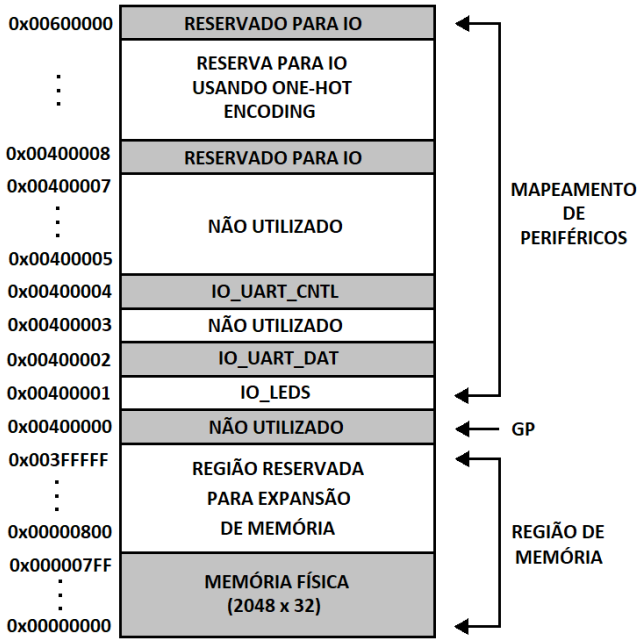
Fonte: Harris e Harris (2021).

De volta ao *FemtoRV*, ele foi inicialmente concebido para ser executado em uma *iCEStick*, uma placa de desenvolvimento da Lattice com um FPGA de apenas 1280 elementos lógicos. A *iCEStick* não conta com muitos periféricos integrados, por isso foram mapeados apenas os LEDs e uma UART como periféricos no projeto original. O método, utilizado por Bruno Levy no mapeamento de I/Os, foi o *one-hot encoding*, que utiliza apenas 1 bit para o endereçamento dos periféricos. Esta escolha aconteceu devido a limitação do número de elementos lógicos do FPGA utilizado (Levy, 2022). A utilização do *one-hot encoding* permite que apenas um periférico seja acessado por vez. A Figura 18 apresenta o mapeamento original para a placa *iCEStick*. Nela é possível ver a memória física e os periféricos mapeados.

Para acessar um periférico através de código, é primordial carregar inicialmente o registrador GP com o endereço do ponto de partida do mapeamento dos periféricos, neste caso, 0x00400000. Em seguida, acrescenta-se um deslocamento para percorrer entre os diferentes endereços a partir do ponto de partida definido por GP. Ao efetuar uma operação de carregamento (*load*) ou armazenamento (*store*) no endereço indicado por GP + deslocamento, estaremos automaticamente interagindo com os periféricos, podendo realizar leituras ou escritas tal como se estivéssemos operando na memória.

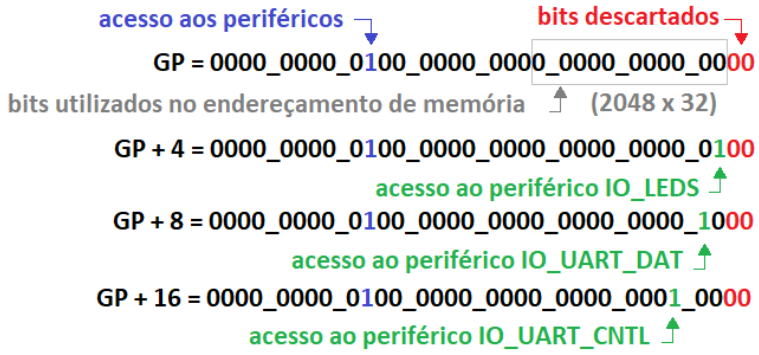
Este tipo de codificação sacrifica alguns endereços em troca da simplicidade de implementação. Mesmo possuindo mais recursos lógicos disponíveis na placa *DE10-Lite*, optamos por manter o mapeamento original devido a forma direta de acesso aos periféricos via código. É importante relembrar que o endereçamento do *FemtoRV* é desalinhado em 2 bits, ou seja, os dois bits menos significativos do barramento de endereços (*mem_addr*) são descartados. A Figura 19 ilustra o conceito de codificação *one-hot encoding* utilizado para acesso aos periféricos.

Figura 18 – Mapeamento de I/O.



Fonte: Elaborado pelo autor.

Figura 19 – *One-hot encoding.*



Fonte: Elaborado pelo autor.

A leitura ou escrita em um periférico, como já dito anteriormente, é realizada por meio das instruções de *load* e *store* em conjunto com o registrador GP que aponta para o início da região dos periféricos. Durante a execução dessas instruções, basta verificar o bit 22 do barramento de endereços (*mem_addr[22]*) para saber se a operação ocorrerá sobre a memória ou em um periférico. Caso a operação ocorra em um periférico, analisa-se qual periférico está sendo requisitado por meio do barramento *mem_wordaddr[periférico]*. A Figura 20 representa um trecho onde essa operação é realizada.

Figura 20 – Operação de *load/store* em I/O e RAM.

```
wire [31:0] RAM_rdata;
wire [29:0] mem_wordaddr = mem_addr[31:2];
wire isIO  = mem_addr[22];
wire isRAM = !isIO;
wire mem_wstrb = |mem_wmask;

always @(posedge clk) begin
   if(isIO & mem_wstrb & mem_wordaddr[IO_LEDS_bit]) begin
  led <= mem_wdata;
   end
end

always @(posedge clk) begin
   if(isIO & mem_wstrb & mem_wordaddr[IO_DISPLAYS_bit]) begin
  display_data <= mem_wdata;
   end
end

wire uart_valid = isIO & mem_wstrb & mem_wordaddr[IO_UART_DAT_bit];
wire uart_ready;

wire [31:0] IO_rdata = mem_wordaddr[IO_UART_CNTL_bit] ? {22'b0, !uart_ready, 9'b0} :
                       mem_wordaddr[IO_SWITCHES_bit]  ? {22'b0, switch}            :
                       mem_wordaddr[IO_KEY_bit]       ? {31'b0, key}               :
                       32'b0;

assign mem_rdata = isRAM ? RAM_rdata : IO_rdata ;
```

Fonte: Trecho de código do módulo *soc.v.*

No âmbito da personalização do *FemtoRV* para o kit *DE10-Lite*, foram introduzidos os botões e interruptores pertencentes ao kit como novos dispositivos de entrada, bem como os displays de 7 segmentos, que foram designados como novos dispositivos de saída, sendo todos eles mapeados em memória. Os LEDs e a UART foram preservados de acordo com o projeto original. O diagrama de blocos com os periféricos disponíveis no kit foi apresentado anteriormente na Figura 6. Nenhum hardware adicional ao kit foi utilizado.

O mesmo conceito de *one-hot encoding*, desenvolvido originalmente, foi utilizado para adicionar os novos periféricos. A Figura 21 apresenta os endereços dos novos periféricos incorporados ao projeto.

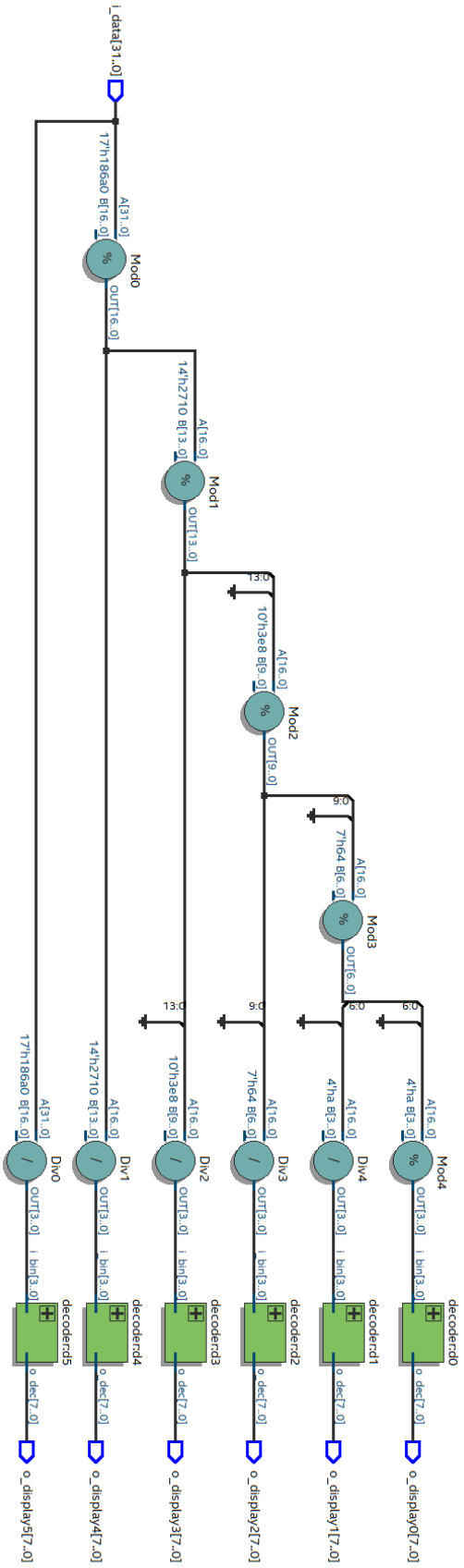
Figura 21 – Endereços dos novos periféricos.

```
// Memory-mapped IO in IO page, 1-hot addressing in word address.
localparam IO_LEDS_bit      = 0;  // W five leds (GP+4)
localparam IO_UART_DAT_bit  = 1;  // W data to send (8 bits) (GP+8)
localparam IO_UART_CNTL_bit = 2;  // R status. bit 9: busy sending (GP+16)
localparam IO_DISPLAYS_bit  = 3;  // Displays de 7-segmentos com decodificador decimal (GP+32)
localparam IO_SWITCHES_bit  = 4;  // Chaves (GP+64)
localparam IO_KEY_bit       = 5;  // Botão (GP+128)
```

Fonte: Trecho de código do módulo *soc.v.*

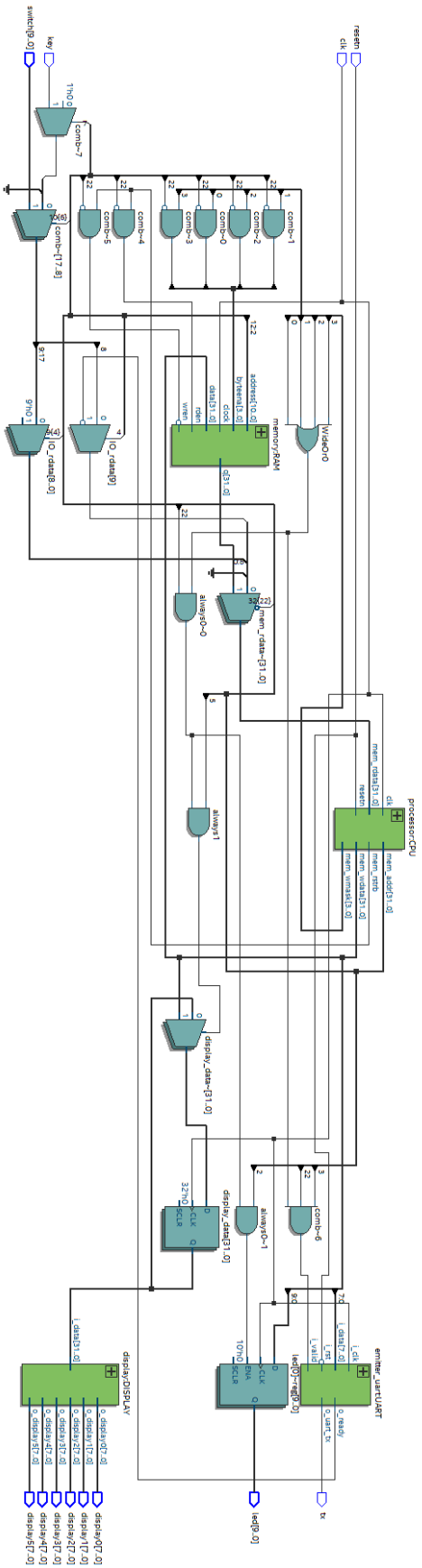
No caso dos displays de sete segmentos, foi implementado um decodificador decimal para converter números binários de 32 bits em representações no formato de unidade, dezena, centena etc. Desta forma, ele opera como um acelerador de hardware, pois não é necessário nenhum processamento posterior via software. Como o kit *DE10-Lite* possui 6 displays de sete segmentos é possível apresentar números inteiros de 0 a 999999. A Figura 22 apresenta o circuito RTL do decodificador.

Figura 22 – Decodificador para display de 7 segmentos.



Fonte: Captura de tela do *Quartus* (*RTL Viewer*).

Figura 23 – Diagrama RTL do *FemtoRV.*



Fonte: Captura de tela do *Quartus* (*RTL Viewer*).

Por último, a Figura 23 ilustra o diagrama geral da implementação da customização do *FemtoRV*, já incluindo os novos periféricos, para o FPGA Intel presente na placa *DE10-Lite*. Na seção superior da figura, estão presentes as entradas de *clock* e *reset* do processador, seguidas dos periféricos KEY e SWITCH[9:0], que foram introduzidos como novos dispositivos de entrada. Além disso, é visível todo o circuito combinacional adicional responsável pelo tratamento dos dados dessas novas entradas. Já na seção inferior, são identificáveis os dispositivos de saída, como a UART (tx), os LEDs[9:0] e os displays de 7 segmentos, juntamente com seus respectivos circuitos.

Nesta implementação, apenas os módulos da CPU e UART foram mantidos conforme o projeto original. Todos os outros circuitos passaram pelo processo de portabilidade e personalização.

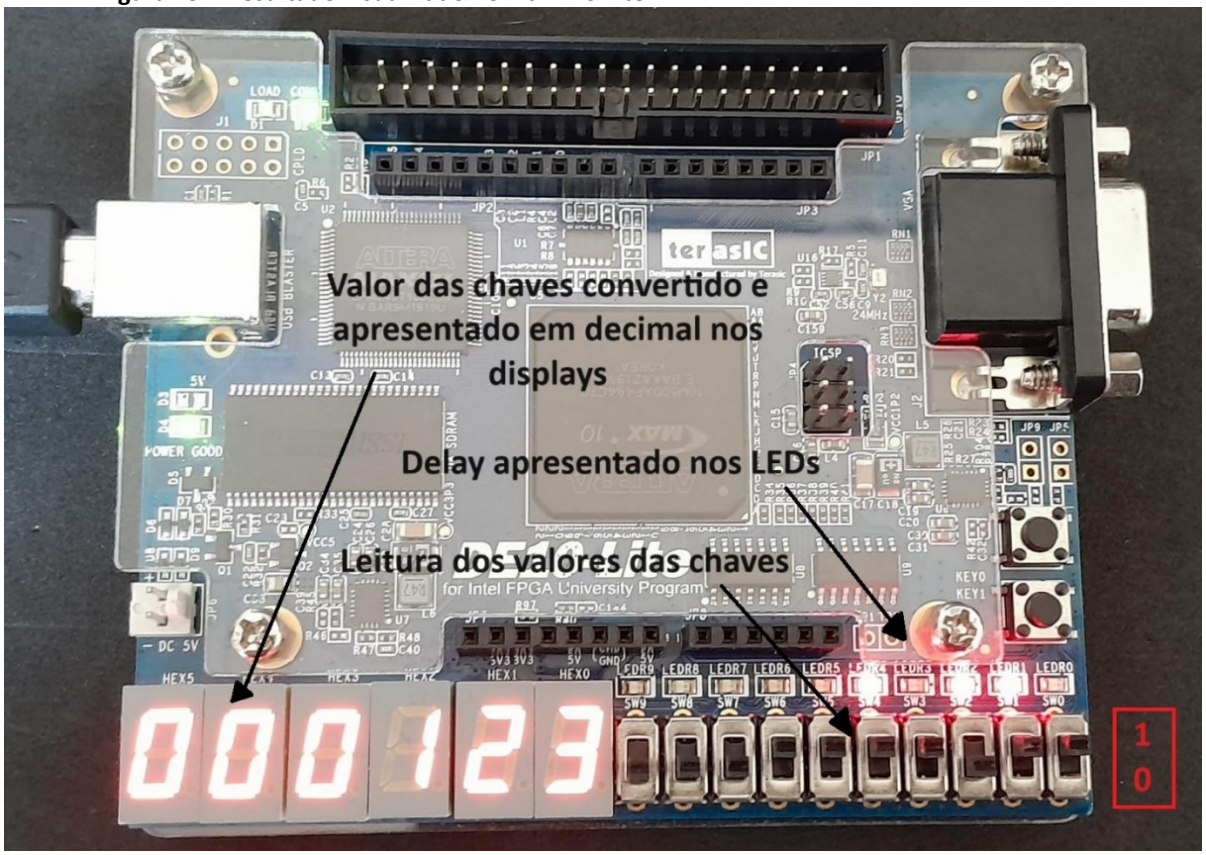
Os periféricos implementados foram testados através de programas escritos em linguagem *assembly* e sintetizados juntamente com o processador. O trecho de código, apresentado na Figura 24, é responsável por ler o estado do conjunto de chaves (SWITCH[9:0]) e, em seguida, exibir o valor correspondente nos displays de 7 segmentos, passando pelo hardware decodificador/acelerador de hardware. Além da funcionalidade principal de leitura das chaves e exibição no display, também foi implementada uma rotina de atraso (*delay*) para melhorar a visualização da execução do programa. O período de atraso é representado nos em binário LEDs na forma de uma contagem progressiva de 0 a 127. A Figura 25 ilustra um dos valores observados no kit durante a execução desse teste.

Figura 24 – Código *assembly* para leitura das chaves e apresentação nos displays.

```
-- Instruções        -- Descrição das Instruções
lui x2 2             -- carrega o valor 0x00002000 nos 20-bits mais significativos do stack pointer (sp)
addi x2 x2 -2048     -- adiciona o valor 0xFFFFF800 ao valor já armazenado no stack pointer
                     -- (define o tamanho da memória em 6k)
lui x3 1024          -- carrega o valor 0x00400000 nos 20-bits mais significativos do global pointer (gp)
addi x3 x3 0         -- adiciona o valor 0x00000000 ao valor já armazenado no global pointer
                     -- (define o endereço inicial dos periféricos)
addi x8 x0 127       -- carrega o valor 0x0000007F no registrador s0
addi x10 x0 0        -- carrega o valor 0x00000000 no registrador a0
sw x10 4(x3)         -- carrega o valor do registrador a0 na posição gp + 4 (acessando diretamente os LEDs)
auipc x6 0           -- call WAIT - carrega o valor 0x00000000 no registrador t1
jalr x1 x6 32        -- call WAIT - carrega o valor t1 + 32 no program counter (pc) e salta 32 bytes para frente
addi x10 x10 1       -- incrementa o valor de a0 em uma unidade (a0 + 1)
bne x10 x8 -16       -- verifica se a0 == 127 e salta 16 bytes para trás de a0 != 127
lw x10 64(x3)        -- lê as chaves (gp + 64) e carrega o valor lido no registrador a0
sw x10 32(x3)        -- escreve o valor de a0 no display (gp + 32)
jal x0 -36           -- salta 36 bytes para trás
ebreak               -- para a execução do programa (o programa nunca chega neste ponto devido o loop infinito)
addi x5 x0 1         -- WAIT - carrega o valor 0x00000001 no registrador t0
slli x5 x5 18        -- desloca o o conteúdo de t0 18-bits para a esquerda (0x00040000)
addi x5 x5 -1        -- WL0 - decrementa o valor de t0 em uma unidade
bne x5 x0 -4         -- verifica se t0 == 0 e salta para 4 bytes para trás se t0 != 0
jalr x0 x1 0         -- retorna para o endereço apontado por x1 (pc = 36)
```

Fonte: Trecho do código switch.mif.

Figura 25 – Resultado visualizado no kit *DE10-Lite*.



Fonte: Elaborado pelo autor.

Os códigos referentes a portabilidade e customização apresentados nesta seção estão disponíveis no repositório *FemtoRV* no *GitHub* do autor e podem ser acessados através do endereço eletrônico https://github.com/diegonagai/FemtoRV.

5 CONCLUSÃO

Neste trabalho, apresentamos o mundo da arquitetura de computadores, onde exploramos o potencial revolucionário da arquitetura RISC-V e sua aplicação prática por meio do *FemtoRV*. Através de um estudo aprofundado, a arquitetura citada foi analisada tanto do ponto de vista de hardware quanto de software.

Na seção 4.1, a microarquitetura minimalista do *FemtoRV* aliada ao seu código-fonte aberto foi explorada. Isso permitiu executar a portabilidade completa do processador de uma plataforma de hardware para outra. Partindo de um FPGA Lattice iCE40 com ferramentas *open-source* para um FPGA Intel MAX10 com as ferramentas oficiais Intel. Essa iniciativa permitiu desbravar os mecanismos de descrição de uma *softcore* e garantiu que o mesmo não ficasse condicionado a um hardware e/ou software específico.

Na seção 4.2, foi dada continuidade ao desenvolvimento do processador. Onde novos periféricos foram criados, perfeitamente integrados seu núcleo e customizados para o kit *DE10-Lite*. Isso permitiu tirar maior proveito do hardware externo ao FPGA e parte do kit de desenvolvimento. Destaca-se nesta seção o desenvolvimento de decodificadores de displays de 7 segmentos por hardware integrado ao *FemtoRV* como um periférico. Onde um valor binário de 32 bits é automaticamente convertido em sinais para acionamento de 6 displays. Permitindo assim a representação de valores ente 0 e 999.999 sem a necessidade de

conversões via software.

Este trabalho deixa bases sólidas para o entendimento do processo de descrição de hardware de alto nível, como a arquitetura de um processador. Trabalhando como uma arquitetura do conjunto de instrução aberta, permitiu aplicar técnicas diferentes no desenvolvimento e avaliar seu desempenho. Além disso, este trabalho tem como legado um arcabouço de hardware sintetizável que pode ser facilmente aplicado em projetos de sistemas embarcados reais.

Como propostas de trabalhos futuros, destacam-se duas possibilidades. A primeira é a criação de novos periféricos e sua integração ao *FemtoRV*, que já foi portado para a placa *DE10-Lite*, seguindo a metodologia apresentada. A segunda possibilidade é a utilização desse núcleo como base para a implementação de novos hardwares para sistemas baseados em tarefas, como proposto por Dantas, De Azevedo e Gimenez (2018).

REFERÊNCIAS

BITTENCOURT, J. **Criando um Projeto no Quartus Prime**. Versão 1.4. Disponível em: https://gcet231.github.io/tut4-fpga-flow/. Acesso em: 12 maio 2023.

BROCK, J. L. **Introduction to logic circuits & logic design with Verilog**. [s.l.] Springer, 2019.

CHU, P. P. **Embedded SoPC design with NIOS II processor and Verilog examples**. [s.l.] John Wiley & Sons, 2012.

DANTAS, L. P.; DE AZEVEDO, R. J.; GIMENEZ, S. P. A novel processor architecture with a hardware microkernel to improve the performance of task-based systems. **IEEE Embedded Systems Letters**, v. 11, n. 2, p. 46–49, 2018.

FLOYD, T. **Sistemas digitais:** fundamentos e aplicações. [s.l.] Bookman Editora, 2009.

GRAPHICS CORPORATION, M. **ModelSim ® User's Manual**. , 1991. Disponível em: www.mentor.com. Acesso em: 12 maio 2023.

HARRIS, S. L.; HARRIS, D. **Digital Design and Computer Architecture, RISC-V Edition**. [s.l.] Morgan Kaufmann, 2021.

IIDA, M. What is an FPGA? **Principles and Structures of FPGAs**, p. 23–45, 2018.

INTEL. **RAM IP Core User Guide**. Disponível em: https://www.intel.com/content/www/us/en/content-details/654455/ram-ip-core-user-guide.html. Acesso em: 12 maio 2023.

INTEL. **Intel Quartus Prime Lite Edition**. Versão 18.1. 2018. Disponível em: https://www.intel.com/content/www/us/en/software-kit/665990/intel-quartus-prime-lite-edition-design-software-version-18-1-for-windows.html?. Acesso em: 12 maio 2023.

LEVY, B. **Learn-FPGA**. Disponível em: https://github.com/BrunoLevy/learn-fpga/. Acesso em: 12 maio. 2023.

NORNBERG, F. **Intel lança nova versão do software Quartus Prime**. Disponível em: https://blog.macnicadhw.com.br/smart-distribution/intel-nova-versao-software-quartus/. Acesso em: 12 maio 2023.

PARAB, J. S.; GAD, R. S.; NAIK, G. M. **Hands-on experience with Altera FPGA development boards**. [s.l.] Springer, 2018.

PATTERSON, D.; WATERMAN, A. **The RISC-V Reader: an open architecture Atlas**. [s.l.] Strawberry Canyon, 2017.

PRADO, A. C. **Tutorial de Modelsim: Verificando o VHDL antes de programar o FPGA**. Disponível em: https://embarcados.com.br/tutorial-de-modelsim-vhdl-fpga/. Acesso em: 12 maio 2023.

RISC-V INTERNATIONAL. **RISC-V Ambassadors**. Disponível em: https://riscv.org/ambassadors/. Acesso em: 12 maio 2023.

TERASIC. **DE10-Lite User Manual**. [s.l: s.n.]. Disponível em: www.terasic.com. Acesso em: 12 maio 2023.

VAKIL, K. **Venus Simulator**. Disponível em: https://venus.cs61c.org/. Acesso em: 12 maio 2023.

AGRADECIMENTOS

Aos professores, aos colegas de curso e a toda equipe da Faculdade SENAI São Paulo, que contribuíram para um excelente curso de pós-graduação mesmo durante o período mais crítico da pandemia de COVID-19. Expresso minha gratidão ao meu orientador pelo suporte fornecido durante o desenvolvimento deste trabalho, bem como aos colegas Eduardo Alvim Guedes Alcoforado e Rani de Almeida Custódio pelas valiosas conversas e discussões sobre o tema. Gostaria de destacar a minha profunda gratidão à minha esposa e à minha querida companheira canina, que me proporcionaram amor, paciência e carinho diários.

SOBRE OS AUTORES

[i] **DIEGO SALVIANO NAGAI**

Engenheiro Eletricista com ênfase em Eletrônica pela Universidade São Judas Tadeu (2013) e cursa a Pós-Graduação em Sistemas Embarcados pela Faculdade de Tecnologia SENAI Anchieta (2022). Tem experiência na área de equipamentos médicos de diagnóstico por imagem nas modalidades tomografia computadorizada e medicina nuclear.

http://lattes.cnpq.br/5530419925531918

[ii] **LEANDRO POLONI DANTAS**

Engenheiro (2004) e Doutor (2018) em Engenharia Elétrica pelo Centro Universitário FEI. Atuou por 15 anos na indústria eletrônica no desenvolvimento de novos produtos. Desde 2009, vem lecionando em cursos de pós-graduação, graduação e de nível técnico em diferentes instituições paulistanas. Atualmente é professor na Faculdade de Tecnologia SENAI e no Insper. https://orcid.org/0000-0003-3674-336X

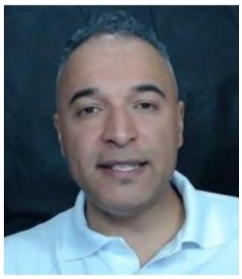
[iii] **MARCONES CLEBER BRITO DA SILVA**

Tecnólogo em Mecatrônica Industrial (2011), Engenheiro Mecatrônico (2013) e Especialista em Engenharia de Manutenção Industrial pela Centro universitário Eniac (2013). Mestre em Tecnologia Nuclear (2020) pela Universidade de São Paulo. Desde 2011, vem lecionando em cursos de nível técnicos e de graduação. Atualmente é professor da Faculdade de Tecnologia SENAI e na FESA. https://orcid.org/0000-0002-3690-1682

[iv] **LUIZ CARLOS CANNO**

Graduado em Tecnologia de Automação Industrial (2009) com Especialização em Gestão Empresarial pela Universidade Nove de Julho (2012), e Especialização em Docência na Educação Profissional e Tecnológica pelo SENAI CETIQT (2015). Professor na Faculdade de Tecnologia SENAI nos cursos graduação e pós-graduação. https://orcid.org/0000-0001-9331-9309

[v] **FERNANDO SIMPLICIO DE SOUSA**



Professor da Faculdade SENAI no curso de Pós-Graduação em Sistemas Embarcados. Mestre em Engenharia Elétrica pela Universidade Federal do ABC (UFABC) e Pós-Graduado (Lato Sensu) pela Universidade Mackenzie. Graduado em Gestão de Pequenas e Médias Empresas pela UNIP e em Projetos Mecânicos pela Faculdade de Tecnologia de São Paulo (UNESP/FATEC-SP). https://orcid.org/0009-0009-5760-4845