



FACULDADE DE TECNOLOGIA SENAI ANCHIETA

PORTEABILIDADE E CUSTOMIZAÇÃO DE UM SOFTCORE RISC-V EM FPGA

PORATABILITY AND CUSTOMIZATION OF A RISC-V SOFTCORE IN FPGA

Diego Salviano Nagai¹,ⁱ
Leandro Poloni Dantas²,ⁱⁱ

Data de submissão: (21/11/2022) Data de aprovação: (dia/mês/ano)

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 evitar o tempo gasto no aprendizado de um novo ambiente de desenvolvimento a cada novo projeto se torna 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. Diante do exposto, este trabalho teve como objetivo abordar a portabilidade de um softcore RISC-V em FPGA passando primeiramente pela revisão de código e simulação do projeto a ser portado e posteriormente adequando o projeto à nova plataforma que embarcará o sistema. Utilizando o kit de desenvolvimento DE10-Lite e o ambiente Intel Quartus Prime Lite Edition foram mantidas as funcionalidades originais do projeto após a portabilidade e novos recursos foram adicionados como o uso de chaves como novos dispositivos de entrada e displays de 7 segmentos como novos dispositivos de saída.

Palavras-chaves: *Softcore, RISC-V, FPGA, Arquitetura de Computadores, Design Digital*

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

¹ Engenheiro Eletricista com ênfase em Eletrônica e aluno da pós-graduação em Sistemas Embarcados na Faculdade de Tecnologia SENAI Anchieta. E-mail: diego_nagai@terra.com.br

² Doutor em Engenharia Elétrica e docente da pós-graduação em Sistemas Embarcados da Faculdade de Tecnologia SENAI Anchieta E-mail: leandro.poloni@sp.senai.br

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 to avoid the time spent learning a new development environment for each new project, it is essential to adopt a portability process. Portability consists of migrating a project developed on one platform or device to another, maintaining the same functionalities. Given the above, this work aimed to address the portability of a RISC-V softcore in FPGA, first going through the code review and simulation of the project to be ported and later adapting the project to the new platform that will embed that system. Using the DE10-Lite development kit and the Intel Quartus Prime Lite Edition environment, the original functionalities of the project were maintained after portability and new features were added such as the use of switches as new input devices and 7-segment displays as new output devices.

Keywords: Softcore, RISC-V, FPGA, Computer Architecture, Digital Design

1 INTRODUÇÃO

Uma 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 das 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 evitar o tempo gasto no aprendizado de um novo ambiente de desenvolvimento a cada novo projeto.

Este trabalho tem como objetivo apresentar a portabilidade de um *softcore* RISC-V desenvolvido originalmente com ferramentas *open-source* para a placa de desenvolvimento iCEStick que possui uma FPGA Lattice iCE40 para uma placa DE10-Lite com uma FPGA Intel MAX10 utilizando as ferramentas oficiais da Intel. Além da portabilidade que mantém as funcionalidades originais do projeto, este trabalho tem como objetivo implementar periféricos customizados especialmente para placa DE10-Lite.

2 DESENVOLVIMENTO

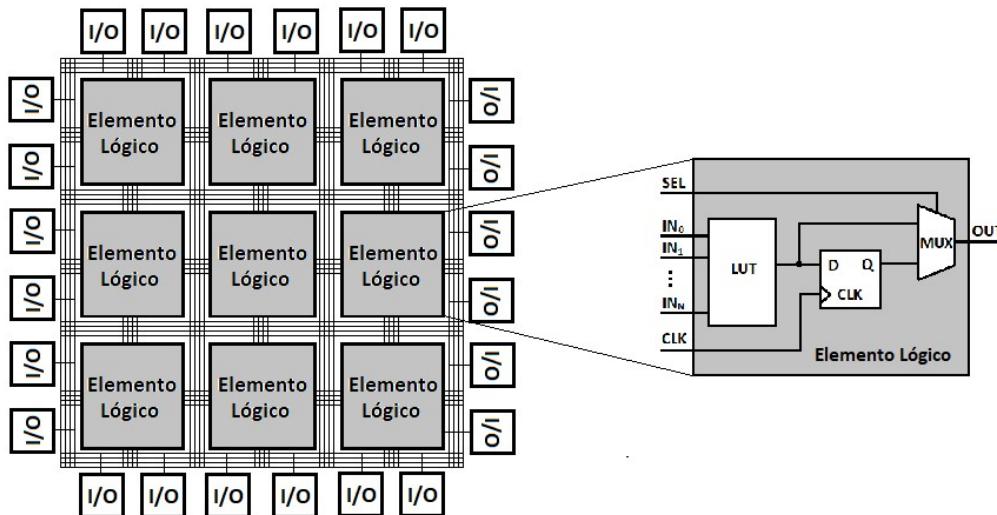
Esta seção apresenta brevemente os principais conceitos que foram necessários para o desenvolvimento deste trabalho. Inicialmente aborda a estrutura típica das FPGAs e apresenta as linguagens de descrição de hardware (HDLs, *hardware description languages*) mais utilizadas. Esta seção também introduz 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

Uma FPGA é um dispositivo lógico programável que pode ser usado para implementar qualquer tipo de circuito digital graças a sua estrutura interna. Segundo (AMANO, 2018), uma 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 uma FPGA.

Figura 1 – Estrutura típica de uma FPGA



Fonte: Autor, 2022

O elemento lógico pode expressar 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 da FPGA com outros dispositivos. Já a matriz de interconexões conecta os diferentes elementos da estrutura interna via roteamento programável (FLOYD, 2007).

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 (AMANO, 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*. Por meio delas é possível descrever a estrutura de qualquer

círculo digital e seu comportamento funcional. As HDLs mais utilizadas são o VHDL e o Verilog (LAMERES, 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 (LAMERES, 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 um 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, 2022).

Embora o ISA descreva todas as especificações tanto para o *hardware* como para o *software*, segundo (HARRIS, 2022) 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 uma 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 em seu ambiente de desenvolvimento seus próprios *softcores*, por exemplo temos o MicroBlaze da Xilinx e o Nios II da Intel que possuem arquiteturas proprietárias (PARAB, 2018), entretanto além dos *softcores* proprietários existem também diversas opções *open-source* e em diferentes arquiteturas, incluindo a arquitetura RISC-V, que tem se tornado a principal escolha no desenvolvimento de *softcores* devido a sua característica modular e *royalty-free*.

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 Califórnia por Krste Asanović, Andrew

Waterman e David Patterson. Suas principais características são a modularidade de um ISA *open-source* e o modelo de licenciamento *royalty-free*.

Segundo (PATTERSON, 2017), no núcleo há um ISA básico, chamado 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 Tabela 1 apresenta o conjunto de instruções RV32I.

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} = rs2 _{7:0}
0100011 (35)	001	-	S	sh rs2, imm(rs1)	store half	[Address] _{15:0} = rs2 _{15:0}
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 << rs2 _{4:0}
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 >> rs2 _{4:0}
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2 _{4:0}
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 imm_{11:0}
- iimm: 5-bit unsigned immediate in imm_{4:0}
- upimm: 20 upper bits of a 32-bit immediate, in imm_{31:12}
- Address: memory address: rs1 + SignExt(imm_{11:0})
- [Address]: data at memory location Address
- BTA: branch target address: PC + SignExt({imm_{12:1}, 1'b0})
- JTA: jump target address: PC + SignExt({imm_{20:1}, 1'b0})
- label: text indicating instruction address
- SignExt: value sign-extended to 32 bits
- ZeroExt: value zero-extended to 32 bits

3 METODOLOGIA

Esta seção apresenta os materiais e métodos utilizados no desenvolvimento deste trabalho. Inicialmente o *softcore* escolhido para o processo de portabilidade tem suas principais características destacadas. Apresenta-se também a placa de desenvolvimento utilizada na implementação bem como seu ambiente de desenvolvimento. Por fim discorre-se sobre a metodologia utilizada nas fases de simulação e síntese do *softcore*.

3.1 Softcore FemtoRV

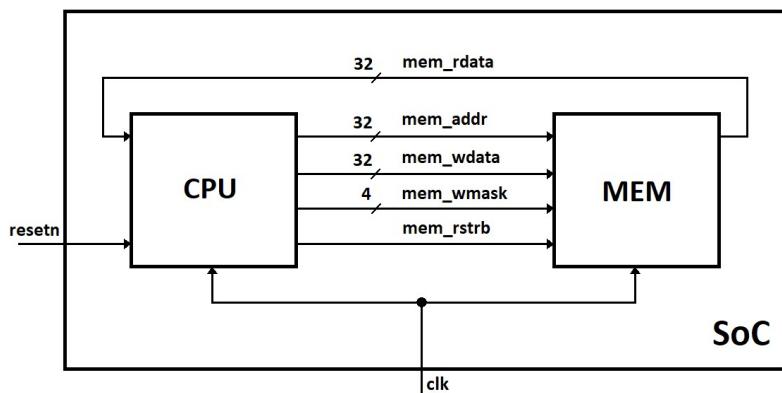
O *softcore* escolhido para o processo de portabilidade e customização foi o FemtoRV, desenvolvido por Bruno Levy, um dos embaixadores da comunidade RISC-V.

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 e hobbista 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 AMBASSADORS, 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 (LEARN-FPGA, 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

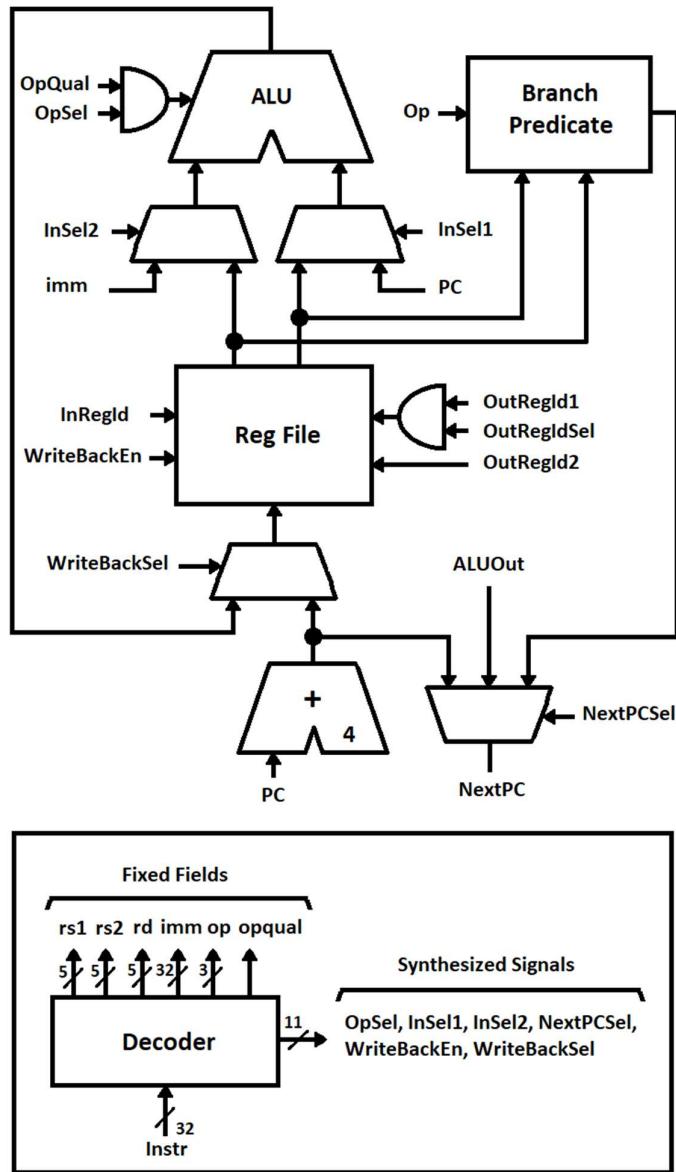


Fonte: AUTOR, 2022

Analizando mais detalhadamente a microarquitetura da **CPU** podemos identificar os principais elementos do *datapath*: o **decoder**, o **register 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 **register file** contém 32 registradores de 32-bits conforme a especificação RISC-V. A **ALU** é responsável pelas operações lógicas e aritméticas entre registradores, além de também realizar operações com imediatos e endereços de salto. O **branch predicate** é responsável pela sinalização do tipo de salto a ser executado, sendo ele condicional ou incondicional. A Figura 3 apresenta o diagrama em blocos da CPU com os elementos do *datapath*.

Figura 3 – Diagrama de blocos da CPU do FemtoRV

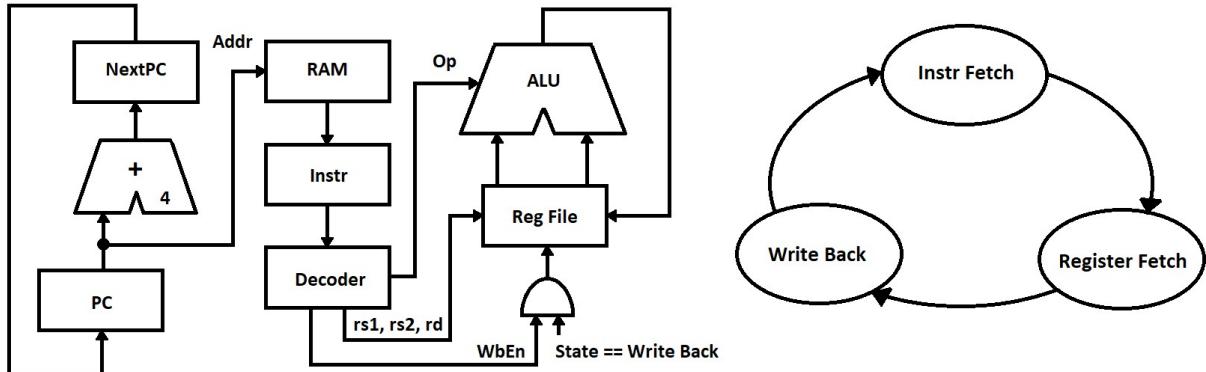


Fonte: LEARN-FPGA, 2022

Os sinais que controlam o fluxo de dados (*datapath*) da Figura 3 são fornecidos por uma **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 fim salva o resultado da operação entre os registradores (*Write Back*). A Figura 4 apresenta o diagrama de estados da unidade de controle.

Figura 4 – Diagrama de estados da unidade de controle

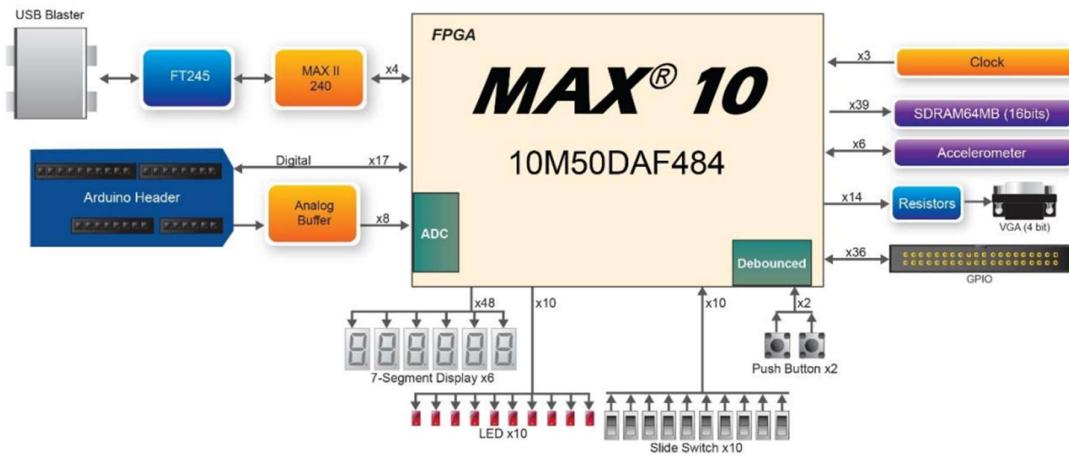


Fonte: LEARN-FPGA, 2022

3.2 Kit de desenvolvimento DE10-Lite

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

Figura 5 – 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 (free). 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 a FPGA ou 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 o Intel Quartus Prime Lite Edition 18.1, que é uma versão gratuita da ferramenta 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, 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 utilizamos 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 (MENTOR GRAPHICS, 2016). 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. A Figura 6 apresenta o fluxo de simulação e síntese utilizado no projeto.

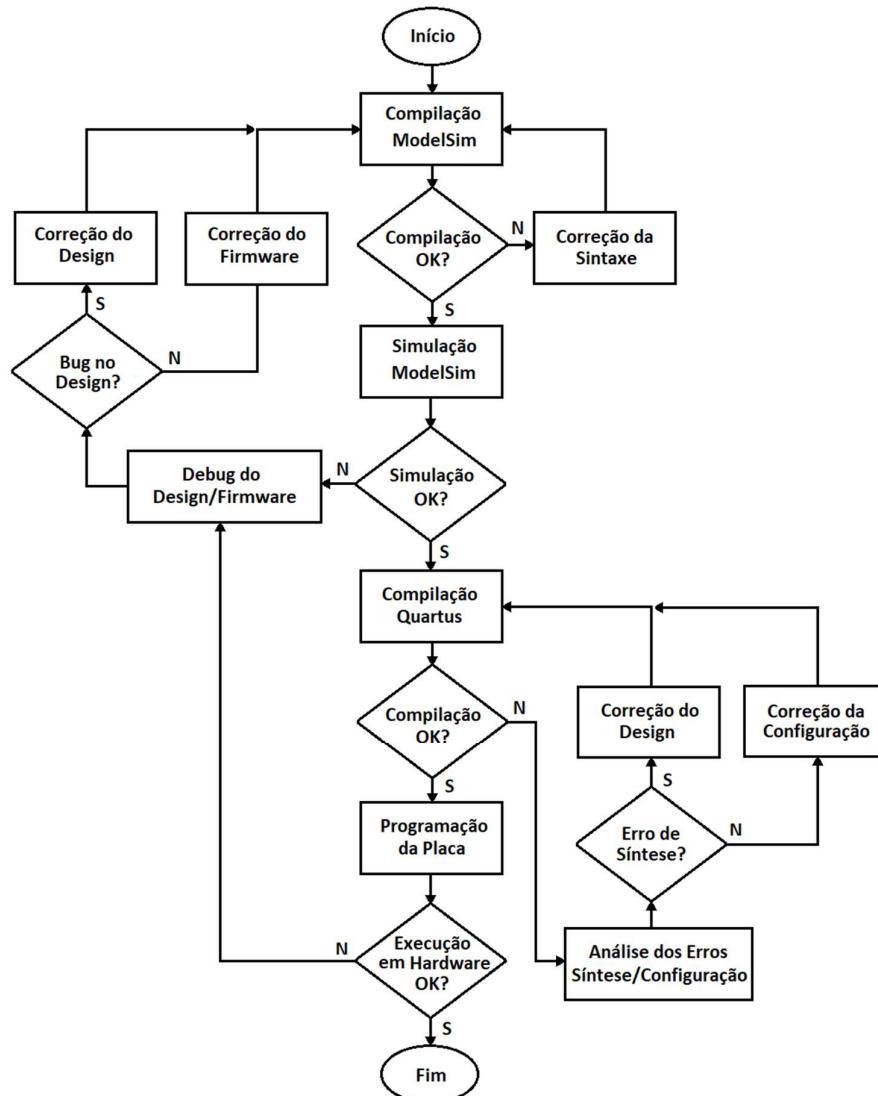
3.6 Síntese

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 da FPGA escolhida. Após o *place and route* acontece a fase de *assembler*, que trabalha na geração do arquivo de programação da FPGA e por fim é realizada uma análise de tempo (*timing*

analysis), onde os tempos de propagação do circuito são calculados após o mapeamento na FPGA (BITTENCOURT, 2021).

Conforme a Figura 6, caso algum erro de compilação ocorra se faz necessário verificar se o problema é relacionado a síntese ou a alguma configuração da 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 da 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 (INTEL, 2022). Com a compilação finalizada com sucesso prossegue-se com a programação da placa e teste 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.

Figura 6 – Fluxo de simulação e síntese



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 também 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 Levy¹. Como já era esperado, na primeira tentativa de simulação diversos erros de compilação foram retornados dentro do ambiente ModelSim devido as diferenças entre os ambientes de desenvolvimento. A Figura 7 apresenta alguns dos erros encontrados durante a compilação do projeto.

Figura 7 – Erros de Compilação no ModelSim

```

Transcript
# Reading D:/intelFPGA_lite/18.1/modelsim_ase/tcl/vsim/pref.tcl
# Loading project step16
# vlog -work work -stats=none D:/step16/step16.v
# Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct 5 2016
# ** Error: ** while parsing file included at D:/step16/step16.v(8)
# ** at clockworks.v(23): Cannot open `include file "../../RTL/PLL/femtopll.v".
# -- Compiling module Clockworks
# -- Compiling module Memory
# ** Error: D:/step16/step16.v(12): (vlog-2892) Net type of 'mem_addr' must be explicitly declared.
# ** Error: D:/step16/step16.v(15): (vlog-2892) Net type of 'mem_wdata' must be explicitly declared.
# ** Error: D:/step16/step16.v(16): (vlog-2892) Net type of 'mem_wmask' must be explicitly declared.
# ** Error: D:/step16/step16.v(11): (vlog-2892) Net type of 'clk' must be explicitly declared.
# ** Error: D:/step16/step16.v(14): (vlog-2892) Net type of 'mem_rstrb' must be explicitly declared.
# -- Compiling module Processor
# ** Error: D:/step16/step16.v(242): (vlog-2730) Undefined variable: 'LOAD_data'.
# ** Error: D:/step16/step16.v(247): (vlog-2730) Undefined variable: 'state'.
# ** Error: D:/step16/step16.v(247): (vlog-2730) Undefined variable: 'EXECUTE'.
# ** Error: D:/step16/step16.v(248): (vlog-2730) Undefined variable: 'WAIT_DATA'.
# ** Error (suppressible): D:/step16/step16.v(273): (vlog-2388) 'LOAD_data' already declared in this scope (Processor).
# ** Error: D:/step16/step16.v(310): 'EXECUTE' already declared in this scope.
# ** Error: D:/step16/step16.v(312): 'WAIT_DATA' already declared in this scope.
# ** Error (suppressible): D:/step16/step16.v(314): (vlog-2388) 'state' already declared in this scope (Processor).
# -- Compiling module Clockworks
# ** Error: D:/step16/step16.v(378): (vlog-2892) Net type of 'LEDS' must be explicitly declared.
# ** Error: D:/step16/step16.v(376): (vlog-2892) Net type of 'CLK' must be explicitly declared.
# ** Error: D:/step16/step16.v(377): (vlog-2892) Net type of 'RESET' must be explicitly declared.
# ** Error: D:/step16/step16.v(379): (vlog-2892) Net type of 'RXD' must be explicitly declared.
# ** Error: D:/step16/step16.v(380): (vlog-2892) Net type of 'TXD' must be explicitly declared.
#
#
# vlog -work work -stats=none D:/step16/clockworks.v
# Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct 5 2016
# ** Error: D:/step16/clockworks.v(23): Cannot open `include file "../../RTL/PLL/femtopll.v".
# -- Compiling module Clockworks
#
#
# vlog -work work -stats=none D:/step16/femtopll.v
# Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct 5 2016
#
#
# vlog -work work -stats=none D:/step16/riscv_assembly.v
# Model Technology ModelSim - Intel FPGA Edition vlog 10.5b Compiler 2016.10 Oct 5 2016
# ** Error: D:/step16/riscv_assembly.v(42): (vlog-2155) Global declarations are illegal in Verilog 2001 syntax.
# ** Error: (vlog-13069) D:/step16/riscv_assembly.v(43): near "initial": syntax error, unexpected initial, expecting class.
#
#
# 4 compiles, 3 failed with 22 errors.

ModelSim>
```

Fonte: Captura de tela do ModelSim

¹ Repositório: https://github.com/BrunoLevy/learn-fpga/tree/master/FemtoRV/TUTORIALS/FROM_BLINKER_TO_RISCV

Para corrigir os erros 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 da FPGA Lattice, desta forma eram incompatíveis com a 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 na incompatibilidade 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. A Figura 8 mostra o uso das *tasks* na inicialização do módulo de memória.

Figura 8 – 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 address 400
    // to address 800
    LI(s1,16);
    LI(s0,0);
    Label(L0_);
    LB(al,s0,400);
    SB(al,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 (VENUS, 2022). A Figura 9 mostra o ambiente do simulador Venus.

Figura 9 – Simulador Venus

The screenshot shows the Venus Simulator interface. At the top, there are tabs: Venus, Editor, Simulator (which is selected), and Chocopy. Below the tabs are control buttons: Run, Step, Prev, Reset, Dump, Trace (which is highlighted), and Re-assemble from Editor.

The main area displays assembly code in two columns: Basic Code and Original Code. The columns are labeled PC, Machine Code, Basic Code, and Original Code. The assembly code includes instructions like addi, li, lb, sb, auipc, jalr, and call WAIT.

To the right of the assembly code is a register dump table. It has columns for Registers, Memory, Cache, and VDB. The Integer (R) tab is selected. The registers listed are zero, ra, sp, gp, tp, t0, t1, t2, and s0, each with its current value displayed in a text input field.

At the bottom left, there are buttons for Copy!, Download!, and Clear!. A console output window is present, showing the message "console output". At the bottom right, there is a "Display Settings" button with a dropdown menu set to "Hex".

PC	Machine Code	Basic Code	Original Code
0x0	0x00000513	addi x10 x0 0	li a0, 0
0x4	0x01000493	addi x9 x0 16	li s1, 16
0x8	0x00000413	addi x8 x0 0	li s0, 0
0xc	0x19040583	lb x11 400(x8)	L0: lb a1, 400, s0
0x10	0x32B40023	sb x11 800(x8)	sb a1, 800, s0
0x14	0x00000317	auipc x6 0	call WAIT
0x18	0x03C300E7	jalr x1 x6 60	call WAIT
0x1c	0x00000317	auipc x6 0	call WAIT
0x20	0x034300E7	jalr x1 x6 52	call WAIT
0x24	0x00140413	addi x8 x8 1	addi s0, s0, 1

Fonte: VENUS, 2022

Todos os programas utilizados neste trabalho foram compilados com a utilização do simulador Venus. A Figura 10 mostra a inicialização do módulo de memória com o código de máquina gerado pelo simulador Venus.

Figura 10 – Inicialização da memória com o código de máquina gerado no simulador Venus

```

initial begin
    MEM[0] = 32'h000000513; // 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'h000000317; // 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'h000000317; // 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'h000008067; // 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

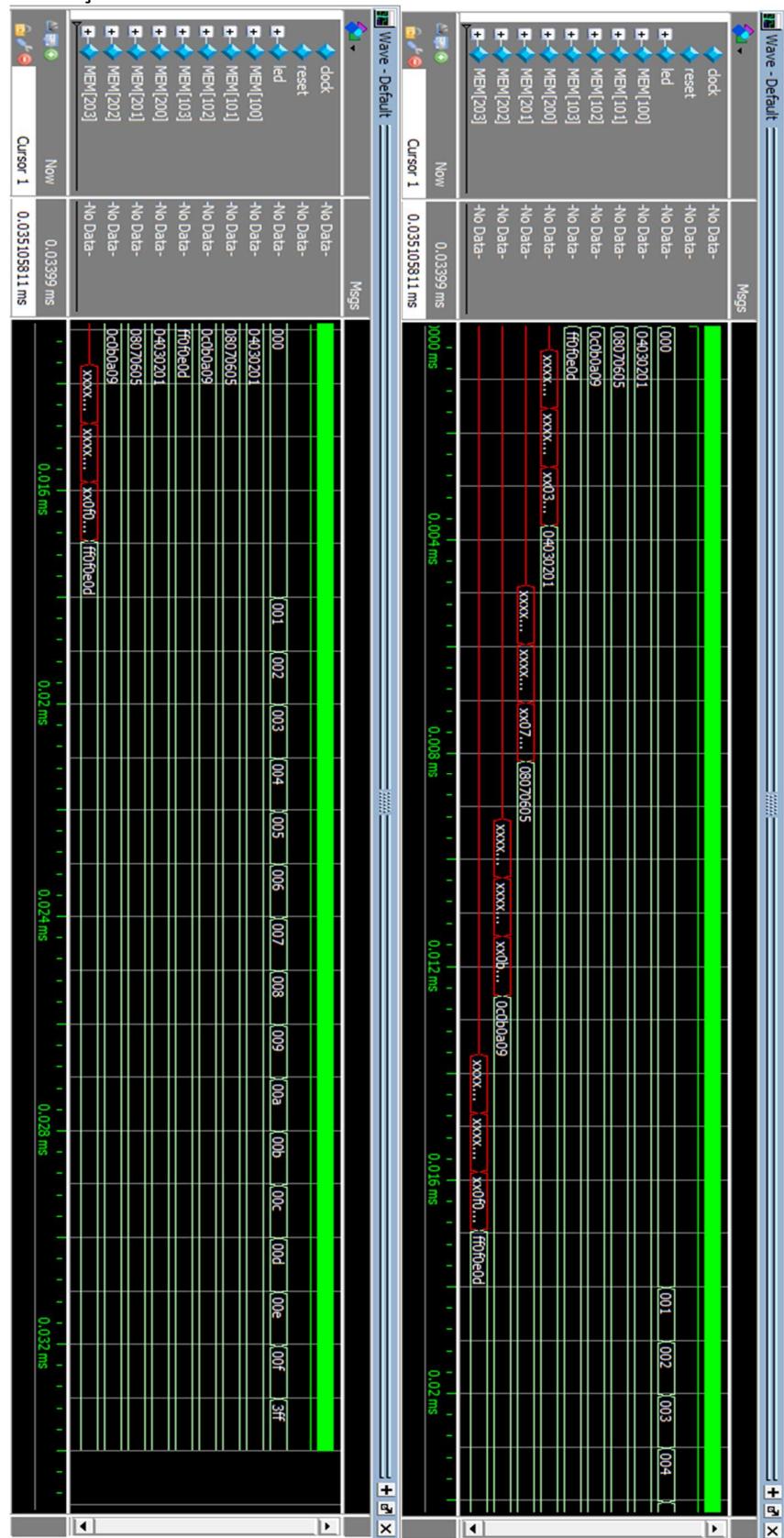
```

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

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 o conteúdo dos endereços 100, 101, 102 e 103 para os endereços 200, 201, 202 e 203 da memória *byte a byte*, 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. A Figura 11 apresenta o resultado da simulação via Waveforms.

Figura 11 – Simulação via Waveforms

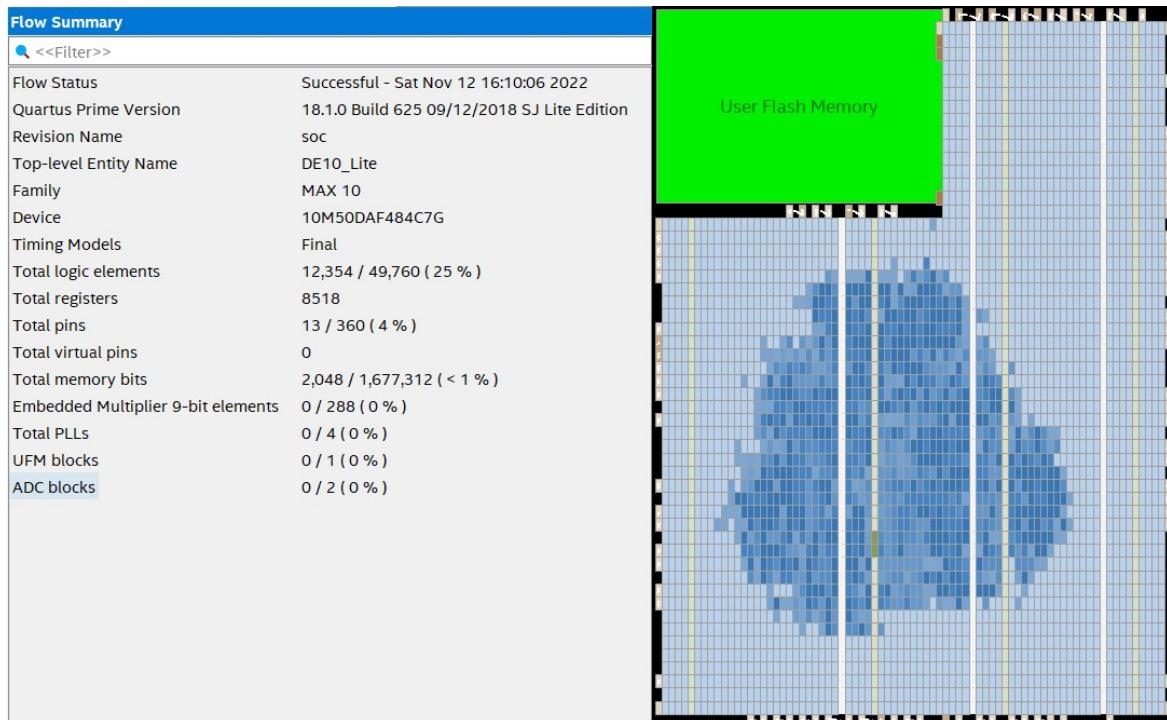


Fonte: Captura de tela do ModelSim (Waveforms)

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*) da FPGA Lattice.

Devido o espalhamento da lógica a execução em *hardware* não funcionou devido problemas de *timing* (tempo de propagação). Além disso a implementação em FF utilizou 25% dos elementos lógicos disponíveis na FPGA MAX10. A Figura 12 apresenta o relatório da síntese e o *place and route* do circuito sintetizado via Chip Planner Viewer.

Figura 12 – Implementação da memória em flip-flops



Fonte: Intel Quartus Prime

Para resolver o problema da implementação da memória foi utilizado o IP RAM-1PORT da Intel que faz uso dos blocos internos de RAM na FPGA (INTEL, 2008). A Figura 13 apresenta a interface gráfica de configuração do IP (*Intellectual Property*), onde é possível determinar as características como largura da palavra e profundidade da memória entre outros.

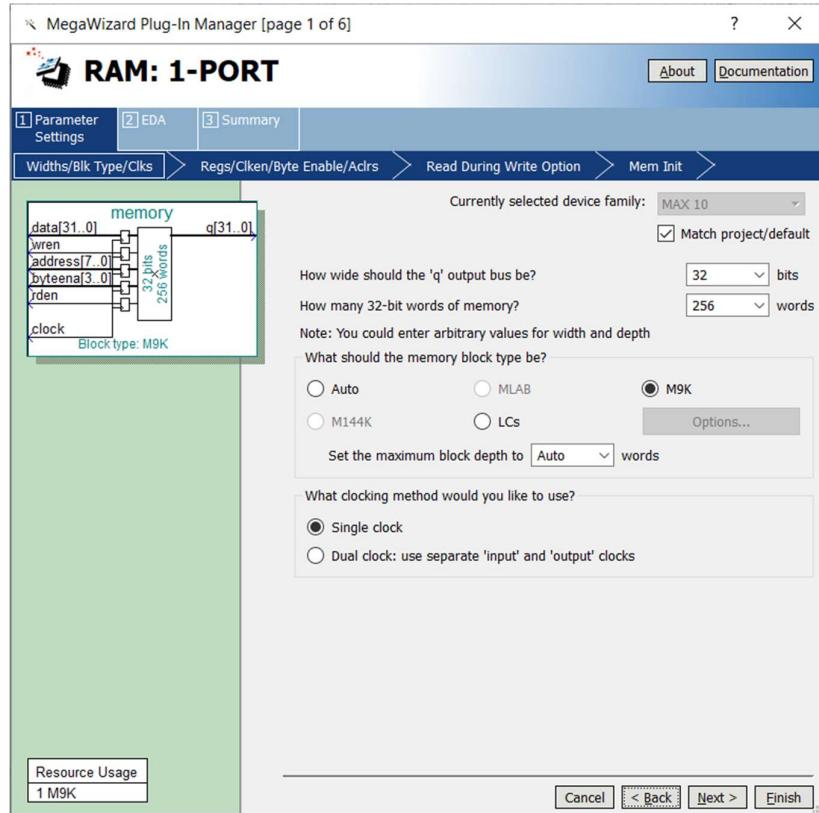
Os IPs são módulos funcionais pré-desenvolvidos que implementam funções específicas e otimizadas. A integração do IP RAM-1PORT com o projeto foi realizada diretamente via instanciação no código em Verilog conforme a Figura 14.

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 pois a memória utilizada tem uma palavra de 32 bits, portanto um endereçamento desalinhado de memória.

Destaca-se que, de acordo com a especificação da ISA RISC-V, o contador de programa (PC) recebe um incremento de 4 bytes para acessar a instrução subsequente. Outro ponto de destaque é em relação aos sinais de habilitação (*enable*) do IP. Como o IP utilizado possui os

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

Figura 13 – Interface gráfica de configuração do IP de RAM



Fonte: Intel Quatus Prime

Figura 14 – 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));

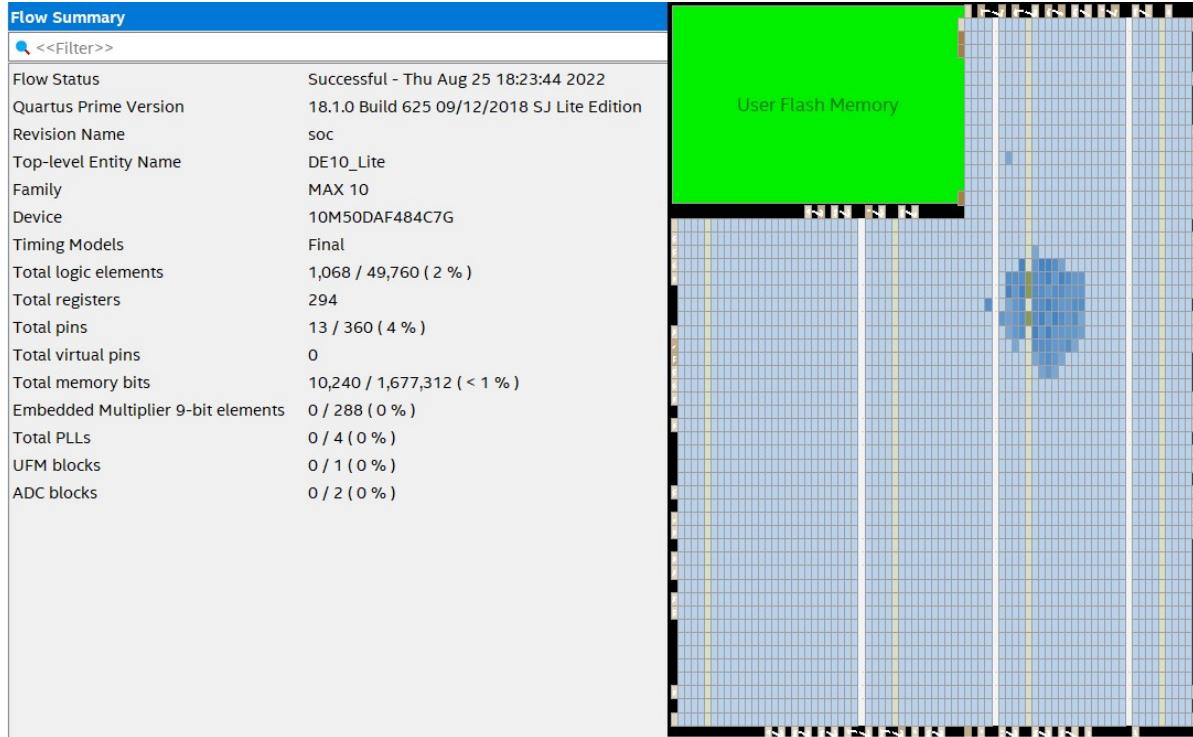
processor CPU(.clk(clock),
              .resetn(reset),
              .mem_addr(mem_addr),
              .mem_rdata(mem_rdata),
              .mem_rstrb(mem_rstrb),
              .mem_wdata(mem_wdata),
              .mem_wmask(mem_wmask),
              .x10(x10));

```

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 15. O uso de BRAM resolveu o problema de *timing* e a execução em *hardware* funcionou exatamente conforme a simulação.

Figura 15 – Implementação da memória em BRAM



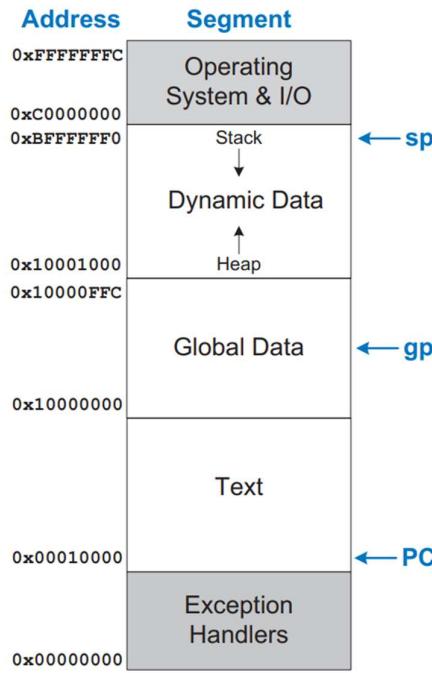
Fonte: Intel Quartus Prime

4.2 Customização

Com a fase de portabilidade finalizada iniciou-se a customização dos periféricos do FemtoRV para a placa DE10-Lite. Como ponto de referência foi utilizado o **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 se baseia na ideia de segmentação das regiões de memória, onde cada região possui um uso específico. A Figura 16 mostra um exemplo de mapeamento comumente utilizado na arquitetura RISC-V.

Figura 16 – Mapeamento de memória



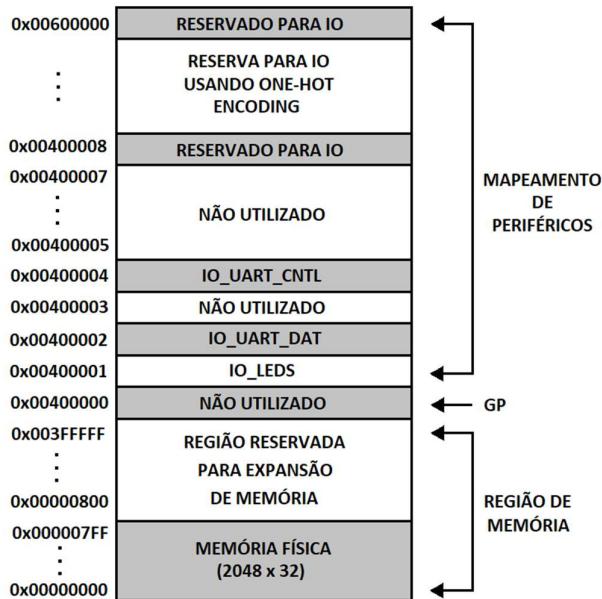
Fonte: HARRIS, 2022

O segmento **Text** armazena o programa em linguagem de máquina e é acessado pelo registrador PC (*program counter*), o segmento **Global Data** armazena as variáveis globais e é acessado pelo registrador GP (*global pointer*), o segmento **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*), a região **Exceptions Handlers** é utilizada para o **boot** de inicialização e também para o tratamento de interrupções, também é acessado pelo GP, a região **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, 2022).

Um ponto importante a se destacar é que as regiões 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.

De volta ao FemtoRV, ele foi inicialmente concebido para ser executado em uma iCEStick, uma placa de desenvolvimento da Lattice com uma 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/O 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 da FPGA utilizada (LEARN-FPGA, 2022). A utilização do *one-hot encoding* permite que apenas um periférico seja acessado por vez. A Figura 17 apresenta o mapeamento original para a placa iCEStick onde é possível ver a memória física e os periféricos mapeados.

Figura 17 – Mapeamento de I/O

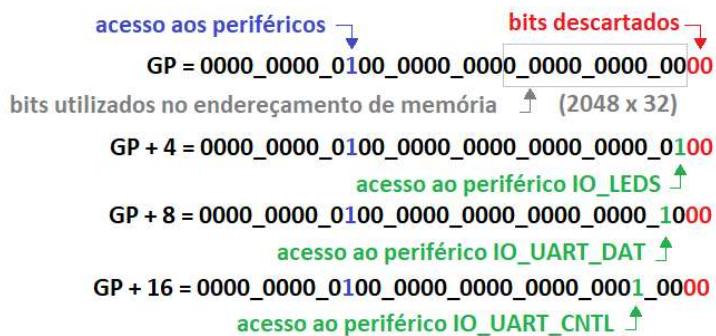


Fonte: AUTOR, 2022

Para acessar um periférico via código primeiramente é necessário carregar o registrador GP com o endereço do início do mapeamento dos periféricos, neste caso 0x00400000, depois adiciona-se um *offset* para a navegação entre os diferentes endereços a partir do ponto inicial apontado por GP. Ao realizar uma operação de *load* ou *store* a no endereço apontado por GP + offset automaticamente acessaremos os periféricos, podendo ler ou escrever como se estivéssemos operando na memória.

Este tipo de codificação sacrifica alguns endereços em troca da simplicidade de implementação. Mesmo possuindo mais recursos lógicos na placa DE10-Lite mantivemos o mapeamento original devido a forma direta que é possível acessar os 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 18 mostra o conceito *one-hot encoding* utilizado no acesso dos periféricos.

Figura 18 – One-hot encoding



Fonte: AUTOR, 2022

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. Na instrução é verificado se o *load/store* será feito em memória ou em um periférico analisando apenas o bit 22 do barramento ***mem_addr[22]***. Caso a operação seja para um periférico analisa-se qual periférico está sendo requisitado por meio do barramento ***mem_wordaddr[periférico]***. A Figura 19 apresenta um trecho de código que demonstra esta operação.

Figura 19 – Operação de *load/store* em I/O / 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***

Como parte da customização do FemtoRV para a placa DE10-Lite foram adicionadas as chaves e botões presentes no kit como os novos dispositivos de entrada e os displays de 7 segmentos como os novos dispositivos de saída mapeados em memória. Os LEDs e a UART foram mantidos conforme o projeto original.

O mesmo conceito do *one-hot encoding* desenvolvido originalmente foi utilizado para adicionar os novos periféricos. A Figura 20 apresenta os endereços dos novos periféricos incorporados ao projeto.

Figura 20 – Endereços dos novos periféricos

```

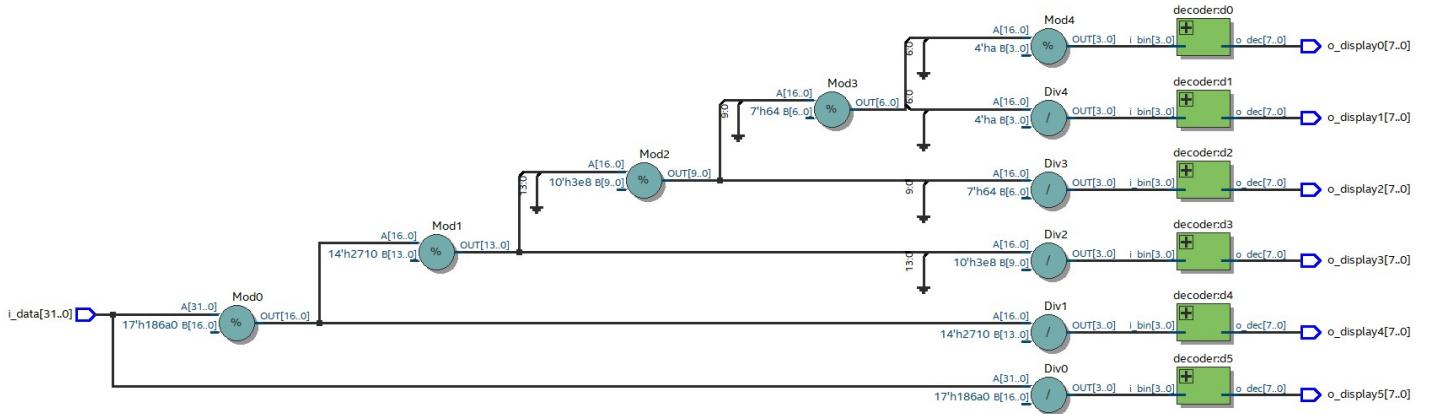
// Memory-mapped IO in IO page, 1-hot addressing in word address.
localparam IO_LEDS_bit      = 0; // W five leds (GP+4)
localparam IO_UART_DAT_bit  = 1; // W data to send (8 bits) (GP+8)
localparam IO_UART_CNTL_bit = 2; // R status. bit 9: busy sending (GP+16)
localparam IO_DISPLAYS_bit  = 3; // Displays 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 7 segmentos também foi implementado um decodificador decimal para converter um número binário de 32 bits em um número decimal. Este módulo ao receber um número de 32 bits faz operações de **módulo** e **divisão** para separar as parcelas decimais do número como unidade, dezena, centena e etc. Como o kit DE10-Lite possui 6 displays de 7 segmentos é possível apresentar números inteiros de **0** a **999999**. A Figura 21 apresenta o circuito decodificador.

Figura 21 – Decodificador para display de 7 segmentos

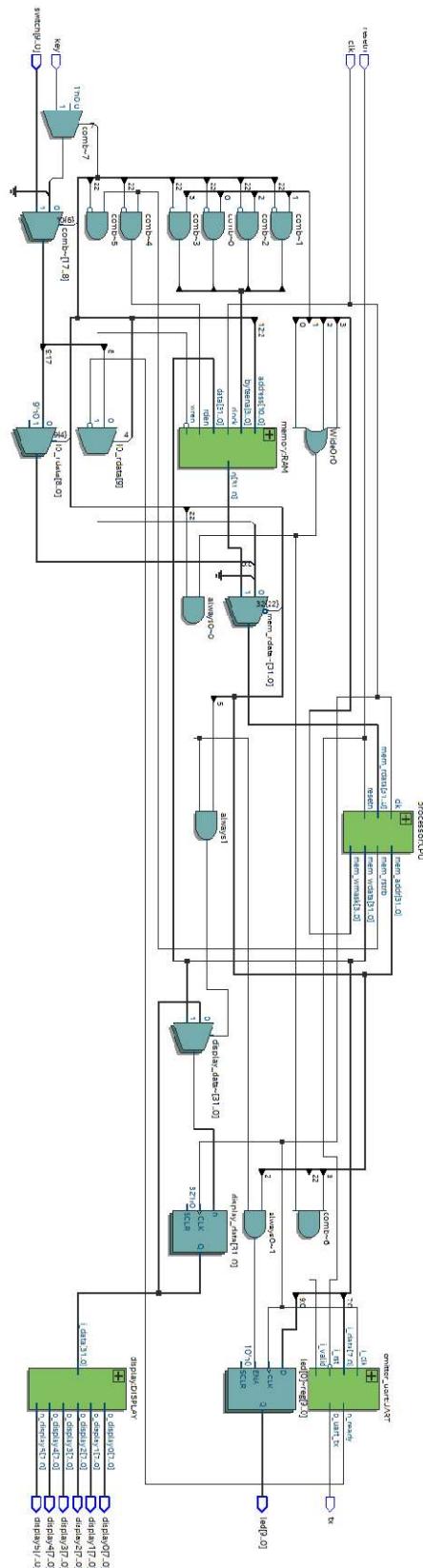


Fonte: Captura de tela do Quatus (RTL Viewer)

Por fim é apresentada a implementação da customização do FemtoRV na placa DE10-Lite. A Figura 22 mostra o diagrama completo do FemtoRV já com os novos periféricos incorporados.

Todos 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>.

Figura 22 – Diagrama RTL do FemtoRV



Fonte: Captura de tela do Quatus (RTL Viewer)

5 CONCLUSÃO

O desenvolvimento deste trabalho favoreceu o estudo e a compreensão da arquitetura RISC-V como um todo, abordando tanto os aspectos de *hardware* quanto de *software*. A microarquitetura minimalista do FemtoRV aliada ao seu código-fonte aberto permitiu sua portabilidade entre plataformas completamente diferentes, partindo de uma FPGA Lattice iCE40 com ferramentas *open-source* para uma FPGA Intel MAX10 com as ferramentas oficiais Intel. Isso permitiu que o estudo deste *softcore* não ficasse condicionado a um *hardware* e/ou *software* específico. Além da portabilidade apresentada manter as funcionalidades originais do projeto, também foram implementadas customizações específicas para a placa DE10-Lite como o uso de chaves como dispositivos de entrada e displays de 7 segmentos como novos dispositivos de saída.

Como proposta de trabalhos futuros, é possível destacar duas possibilidades. A criações de novos periféricos e sua integração ao FemtoRV, já portado para a placa DE10-Lite, segundo a metodologia apresentada. E a utilização do núcleo portado como um *custom design* na plataforma Intel Pathfinder (INTEL, 2022). Esta plataforma permite o desenvolvimento de *software* em linguagem C/C++ dentro de uma IDE baseada no Eclipse.

REFERÊNCIAS

- AMANO, H. **Principles and Structures of FPGAs**. Singapore: Springer Nature, 2018. ISBN 978-981-13-0824-6.
- BITTENCOURT, J. Criando um Projeto no Quartus Prime. **GitHub**, 2021. Disponível em: <<https://gcet231.github.io/tut4-fpga-flow/>>. Acesso em: 15 de out. 2022.
- CHU, P. P. **Embedded SOPC Design with NIOS II Processor and Verilog Examples**. Hoboken: John Wiley & Sons, Inc., 2012. ISBN ISBN 978-1-118-01103-4.
- FLOYD, T. L. **Sistemas Digitais – Fundamentos e Aplicações**. Tradução de NASCIMENTO, JOSÉ L. Porto Alegre, RS: Bookman, 2007. ISBN 0131946099.
- HARRIS, S. L.; HARRIS, D. M. **Digital Design and Computer Architecture - RISC-V Edition**. Cambridge, MA: Morgan Kaufmann, 2022. ISBN 978-0-12-820064-3.
- INTEL. Intel Pathfinder for RISC-V. **Intel**, 2022. Disponível em: <<https://pathfinder.intel.com>>. Acesso em: 19 de nov. de 2022.
- INTEL. Quartus Prime Pro Edition Help Version 18.1 . **Intel**, 2022. Disponível em: <<https://www.intel.com/content/www/us/en/programmable/quartushelp/18.1/index.htm>>. Acesso em: 15 de out. 2022.
- INTEL. RAM IP Core User Guide. **Intel**, 2008. Disponível em: <<https://www.intel.com/content/www/us/en/content-details/654455/ram-ip-core-user-guide.html>>. Acesso em: 22 de out. de 2022.

LAMERES, B. J. **Introduction to Logic Circuits & Logic Design with Verilog**. Switzerland, AG: Springer Nature, 2019. ISBN 978-3-030-13604-8.

LEARN-FPGA. **GitHub**, 2022. Disponível em: <<https://github.com/BrunoLevy/learn-fpga/>>. Acesso em: 01 de out. de 2022.

MENTOR GRAPHICS. **ModelSim User's Manual – Software Version 10.5b**, 2016.

NISSAM, N.; SCHOCKEN, S. **The Elements of Computing Systems**. 2a. ed. Cambridge: The MIT Press, 2021. ISBN 9780262539807.

NORNBERG, F. Intel lança nova versão do software Quartus Prime. **Macnica DHW**, 2020. Disponível em: <<https://blog.macnicadhw.com.br/smart-distribution/intel-nova-versao-software-quartus/>>. Acesso em: 08 de out. de 2022.

PARAB, J. S.; GAD, R. S.; NAIK, G. M. **Hands-on Experience with Altera FPGA Development Boards**. India: Springer Nature, 2018. ISBN 978-81-322-3767-9.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design - RISC-V Edition**. 2a. ed. Cambridge: Morgan Kaufmann, 2021. ISBN 978-0-12-820331-6.

PATTERSON, D.; WATERMAN, A. S. **Guia Prático RISC-V: Atlas de uma arquitetura aberta**. 1a. ed. San Francisco: Strawberry Canyon LLC, 2017. ISBN 978-0-9992491-1-6.

PRADO, A. C. Tutorial de Modelsim: Verificando o VHDL antes de programar o FPGA. **Embarcados**, 2014. Disponível em: <<https://embarcados.com.br/tutorial-de-modelsim-vhdl-fpga/>>. Acesso em: 08 de out. 2022.

RISC-V AMBASSADORS. **RISCV International**, 2022. Disponível em: <<https://riscv.org/ambassadors/>>. Acesso em: 01 de out. de 2022.

TERASIC. **DE10-Lite User Manual**, 2020.

VENUS. **Venus Simulator**, 2022. Disponível em: <<https://venus.cs61c.org>>. Acesso em: 22 de out. 2022.

AGRADECIMENTOS

Aos professores e colegas de curso, que contribuíram para um excelente curso de pós-graduação mesmo que a distância.

A toda equipe da Faculdade de Tecnologia SENAI Anchieta pelo suporte durante o período mais crítico da pandemia de COVID-19.

Ao meu orientador pelo suporte durante o desenvolvimento deste trabalho e aos colegas Eduardo Alvim Guedes Alcoforado e Rani de Almeida Custódio pelas conversas e discussões sobre o tema.

Agradecimentos especiais à minha esposa e minha filha canina, por todo amor, paciência e carinho diário.

SOBRE OS AUTORES

ⁱ **Diego Salviano Nagai**



Possui graduação em Engenharia Elétrica 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 e possui interesse nos temas: Sistemas Digitais, Lógica Programável, Arquitetura de Computadores, Aceleradores de Hardware e Processamento Digital de Sinais. Atualmente trabalha como engenheiro de campo na empresa Philips onde é responsável pelo suporte técnico a equipamentos de diagnóstico por imagem em clínicas, hospitais e instituições de ensino na área de saúde.

ⁱⁱ **Leandro Poloni Dantas**



Doutor em Engenharia Elétrica pelo Centro Universitário FEI, cuja tese apresenta proposta de arquiteturas alternativas para projetos de processadores e microcontroladores com foco em sistemas de tempo real. Possui graduação em Engenharia Elétrica com ênfase em Eletrônica (2004) e mestrado em Dispositivos Eletrônicos Integradas (2008) ambos pelo Centro Universitário FEI. Dissertou sobre estudo de distorção harmônica em transistores MOSFET de porta circular. Atuou por 15 anos na indústria eletrônica no desenvolvimento de novos produtos para diferentes segmentos, com foco principal no projeto de equipamentos eletrônicos para o mercado de segurança eletrônica e automação predial. Desde 2009, vem lecionando em cursos de pós-graduação, graduação e de nível técnico em diferentes instituições paulistanas. Publicou livros sobre eletrônica digital e microcontroladores. (Texto extraído na íntegra da plataforma Lattes). Currículo Lattes: <http://lattes.cnpq.br/6255986062207024>