Departamento de Ciências da Computação (CIC/UnB)
Universidade de Brasília (UnB)

ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES

PROJETO FINAL

IMPLEMENTAÇÃO DE UM PROCESSADOR UNICICLO BASEADO NA ARQUITETURA RISC-V

Número	**Nome completo**
211038217	Caio César Gonçalves Ribeiro
212008894	João Victor Cavallin Pereira

Prof. Ricardo Pezzuol Jacobi

Turma 3

Brasília, 24 de Fevereiro de 2025

Índice

1 Resumo

Nesse projeto, foi desenvolvido, em linguagem de descrição de Hardware, um processador uniciclo baseado na arquitetura RISC-V. Foram utilzados conceitos de lógica digital e arquitetura de processadores para implementar diversos tipos de instruções, que foram testados utilizando código em linguagem de máquina , ,produzido a partir do simulador RARS.

2 Introdução

2.1 Objetivos

Desenvolver Datapath e Controle de um processador RISC-V, para simulação e possivelmente sintetização para um circuito real, com código real.

2.2 Arquitetura e Componentes de Fluxo

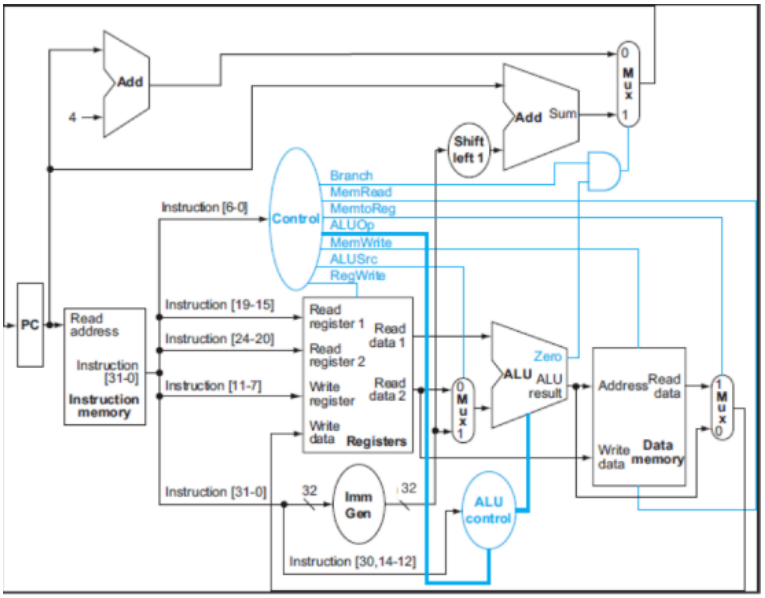


Figura 1: Arquitetura Base

Como mostrado no diagrama acima, diversos módulos são necessários para a implementação do processador.

2.2.1 Contador de Programa (PC)

O PC é um registrador de 32 bits que contém o endereço ROM da instrução a ser executada. No módulo desenvolvido, além de entrada de dados e clock, há uma entrada de reset síncrono para configuração inicial.

2.2.2 ROM

A ROM é uma memória não-volátil que será carregada com as instruções do programa. A instrução localizada pelo endereço na entrada será lida.

2.2.3 Banco de Registradores

São 32 registradores disponíveis para leitura asssíncrona e escrita síncrona, com exceção do registrador x0, que está permanentemente zerado. Há entradas para seleção de dois registradores a serem lidos, para seleção de registrador a ser escrito, e o dado a ser escrito. A habilitação da escrita é controlada, além do clock, por um sinal de controle acionado conforme a necessidade de instrução.

2.2.4 ULA

A Unidade Lógico-Aritmética é controlada por meio de OpCodes próprios gerados a partir de sua unidade de controle. As operações oferecidas são:

Operação	Significado	OpCode
ADD A, B	Z recebe a soma das entradas A, B incluindo o vem-um	0000
SUB A, B	Z recebe A - B	0001
AND A, B	Z recebe a operação lógica A and B, bit a bit	0010
OR A, B	Z recebe a operação lógica A or B, bit a bit	0011
XOR A, B	Z recebe a operação lógica A xor B, bit a bit	0100
SLL A, B	Z recebe a entrada A deslocada B bits à esquerda	0101
SRL A, B	Z recebe a entrada A deslocada B bits à direita sem sinal	0110
SRA A, B	Z recebe a entrada A deslocada B bits à direita com sinal	0111
SLT A, B	Z = 1 se A < B, com sinal	1000
SLTU A, B	Z = 1 se A < B, sem sinal	1001
SGE A, B	Z = 1 se A ≥ B, com sinal	1010
SGEU A, B	Z = 1 se A ≥ B, sem sinal	1011
SEQ A, B	Z = 1 se A == B	1100
SNE A, B	Z = 1 se A != B	1101

Figura 2: Tabela ULA

O módulo também conta com uma saída de Flag para instruções de comparação, e que serão utilizados pelo sub-circuito de Branch e Jump.

2.2.5 RAM

Neste projeto, a RAM pode ser escrita e lida como palavra de 32 bits ou como Byte. Diferentmente do diagrama, a leitura oferece modo com ou sem sinal. Além disso, a leitura é feita de modo assíncrono e não há sinal de controle habilitando a leitura, apenas a escrita.

2.2.6 Gerador de Imediatos

Instruções com imediato carregam consigo, além da representação deste, o modo como deve ser interpretado para leitura pelo processador. O Gerador de Imediatos é o responsável por essa decodificação.

2.3 Adições ao Modelo

O esquemático acima não suporta as instruções JAL, JALR, AUIPC, LUI, acesso a memória em modo de byte ou sem sinal. Dessa forma, faz-se necessário adição de sinais e módulos.

2.3.1 Contador de Programa

Agora a entrada do Contador de Programa contém um multiplexador, controlado por um sinal sJALR, que seleciona a entrada anterior - saída do multiplexador de PC+4 ou Branch - em LOW e a saída da ULA principal em HIGH. Em conjunto com outras mudanças, permite JALR.

2.3.2 Banco de Registradores

Para a entrada de Dados, um multiplexador 4x1, controlado por um sinal de 2 bits RegSrc, seleciona três fontes diferentes: O dado proveniente do estágio de Write-Back, o imediato gerado e o PC+4. Habilita, respectivamente, LUI, JAL e JALR.

2.3.3 ULA

Agora ambas as entradas da ULA possuem um multiplexador 2x1, controlados por ALUSrc1 e ALUSrc2. O primeiro multiplexador seleciona a leitura do conteúdo de registrador em LOW ou PC em HIGH. Permite a implementação de AUIPC.

2.3.4 Lógica de Branch

Agora, além da condição de (Branch AND ALUZero) para desvio, há um desvio incondicional com JAL, de modo que o multiplexador de seleção de (PC+4 ou Branch) passa a ser controlado por S = ((Branch AND ALUZero) + isJAL), sendo isJAL um sinal de controle.

2.3.5 RAM

A RAM tem duas entradas de Flag adicionais para leitura/escrita em Word ou Byte, ou leitura com/sem sinal.

Observando o código das instruções de carregamento e armazenamento da memória, percebe-se que: manejando bytes, o décimo-quarto bit da instrução está em LOW, manejando leitura com sinal, o bit 13 está em LOW. Dessa forma, a lógica para ativação é:

$$ByteEnable = NOT(INSTR(14)) \tag{1}$$

$$SgnEnable = NOT(INSTR(13)) \tag{2}$$

3 Controle

3.1 Controle Principal

No controle principal, a lógica de sinais, não otimizada e desconsiderando alguns termos Dont-Care, é:

TipoInstr	Br	Mem2Reg	MemWr	RegWr	ALU1	ALU2	JALs	JALRs	ALUOP	RegSrc
AUIPC	0	0	0	1	1	1	0	0	00	00
LUI	0	0	0	1	0	0	0	0	00	01
JAL	0	0	0	1	0	1	1	0	00	10
JALR	0	0	0	1	0	1	0	1	00	10
BRANCH	1	0	0	0	0	0	0	0	01	00
LOAD	0	1	0	1	0	1	0	0	00	00
STORE	0	0	1	0	0	1	0	0	00	00
R-TYPE	0	0	0	1	0	0	0	0	10	00
I-TYPE	0	0	0	1	0	1	0	0	11	00

3.2 Controle da ULA

Para as instruções implementadas, além do ALUOp, deve-se diferenciar as instruções pelos campos funct3 e funct7, o último sendo necessário apenas o trigésimo bit da instrução para diferenciação.

ALUOP	funct3	Inst(30)	ALUOpcode	Op
00	x	x	0000	ADD
01	000	x	1100	SEQ
01	001	x	1101	SNE
10	111	x	0010	AND
10	010	x	1000	SLT
10	110	x	0011	OR
10	100	x	0100	XOR
10	000	0	0000	ADD
10	000	1	0001	SUB
10	001	x	0101	SLL
10	101	0	0110	SRL
10	101	1	0111	SRA
11	000	x	0000	ADD
11	111	x	0010	AND
11	110	x	0011	OR
11	100	x	0100	XOR

3.3 Esquemático Final

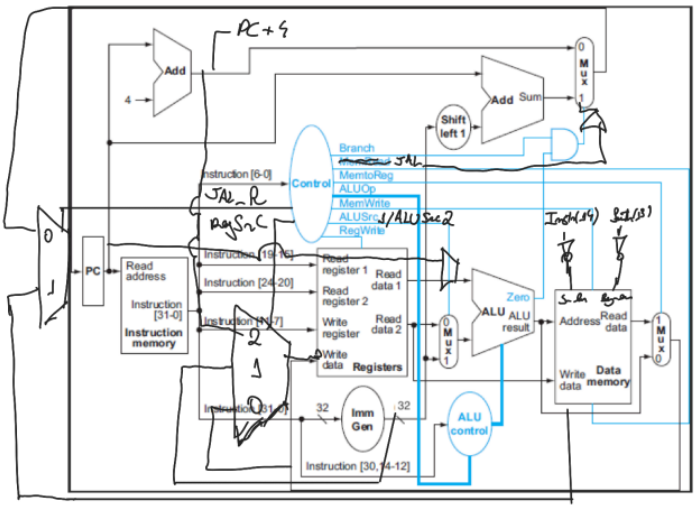


Figura 3: Tabela ULA

4 Testes

4.1 Contador de Programa - PC

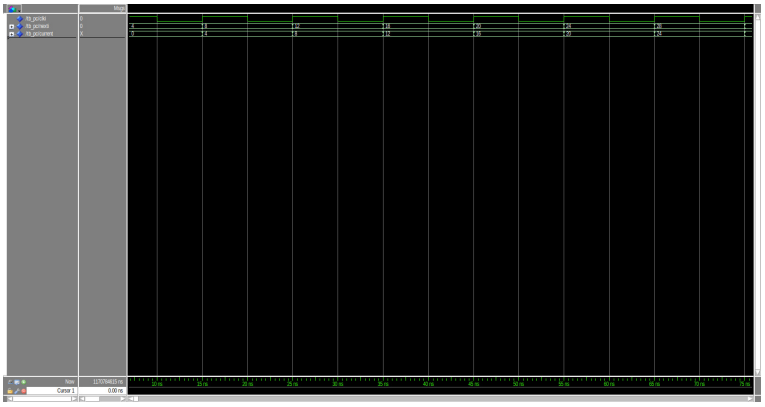


Figura 4: Testbench modulo PC (Wave)

Os sinais registrados no teste de onda representam o próximo valor de pc e o valor atual respectivamente.

4.2 Banco de Registradores - XREGS

4.2.1 Escrita habilitada (wren = '1')

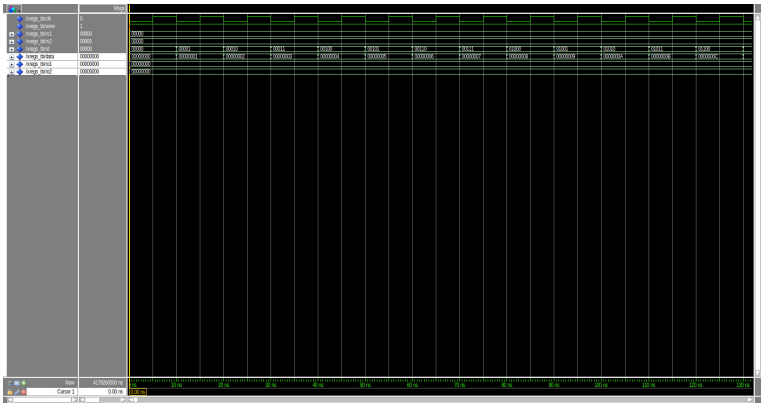


Figura 5: Testbench módulo XREGS (escrita habilitada)

Na leitura do sinal em onda, é percepctivel a escrita em cada registrador, para facilitar os testes e a análise dos resultados foi determinado que cada registrador iria receber como dado seu próprio número. Dessa forma o registrador 1 recebeu o dado 1, o registrador 2 recebeu o dado 2 e assim por diante.

4.2.2 Leitura habilitada (wren = '0')

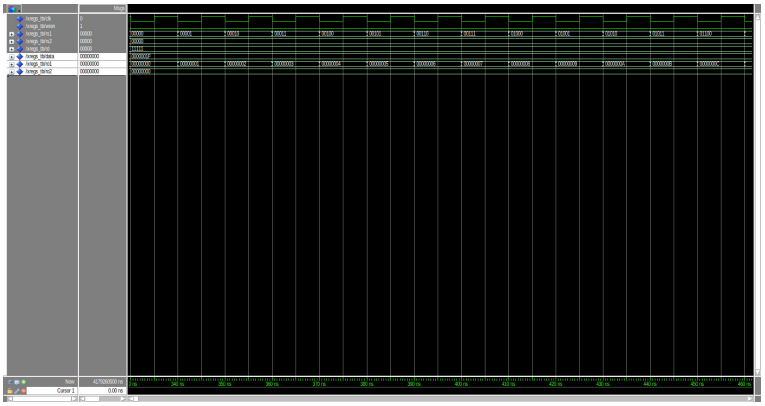


Figura 6: Testbench módulo XREGS (Leitura habilitada)

Nesse próximo teste, percebemos que os dados lidos correspondem aos testes realizados anteriormente.

4.2.3 Registrador ZERO

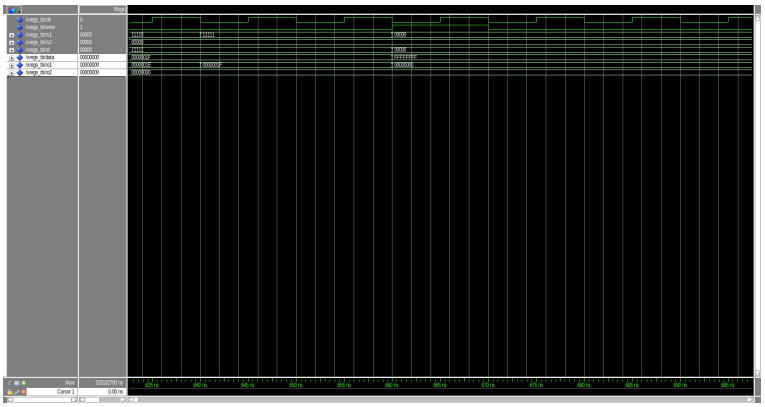


Figura 7: Escrita e leitura do registrador zero

Podemos perceber dois cénarios e apenas um resultado. Mesmo com o modo de escrita habilitado (wren), o registrador zero não altera de valor, mantendo seu valor constante igual a 0.

4.3 Memória RAM

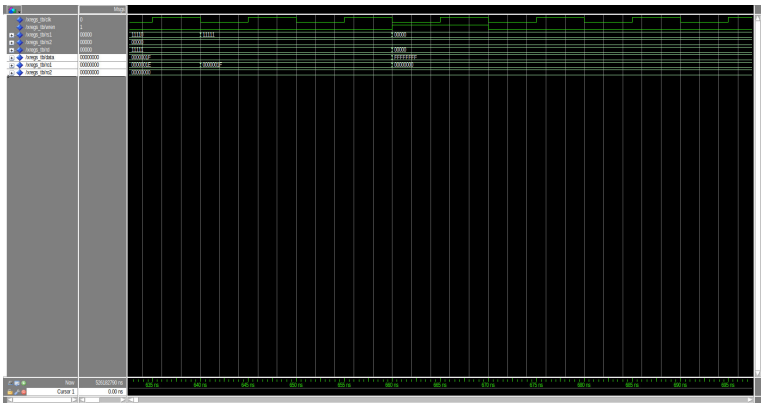


Figura 8: Testbench memória RAM

4.4 Demais módulos

Os demais módulos possuem testes realizados pela ferramenta de asserts, portanto podem ser vistos através dos próprios testbenchs.

4.5 Teste de Código

4.5.1 Operações na Memória, Aritméticas e JAL

Abaixo seguem telas de código, memória e banco de registradores no RARS e no ModelSim, deixando claro a identidade entre estes.

Code	Basic
0x00500513	addi x10,x0,5
0x00000593	addi x11,x0,0
0x00100613	addi x12,x0,1
0x000026b7	lui x13,2
0x00068693	addi x13,x13,0
0xffe00713	addi x14,x0,0xfffffffe
0x004000ef	jal x1,0x00000004
0x00050863	beq x10,x0,0x00000010
0xfff50513	addi x10,x10,0xffffffff
0x00c585b3	add x11,x11,x12
0xff5ff0ef	jal x1,0xfffffff4
0x00b6a023	sw x11,0(x13)
0x00e680a3	sb x14,1(x13)
0x00068403	lb x8,0(x13)
0x0016c483	lbu x9,1(x13)
0x0006a903	lw x18,0(x13)

Figura 9: Caption

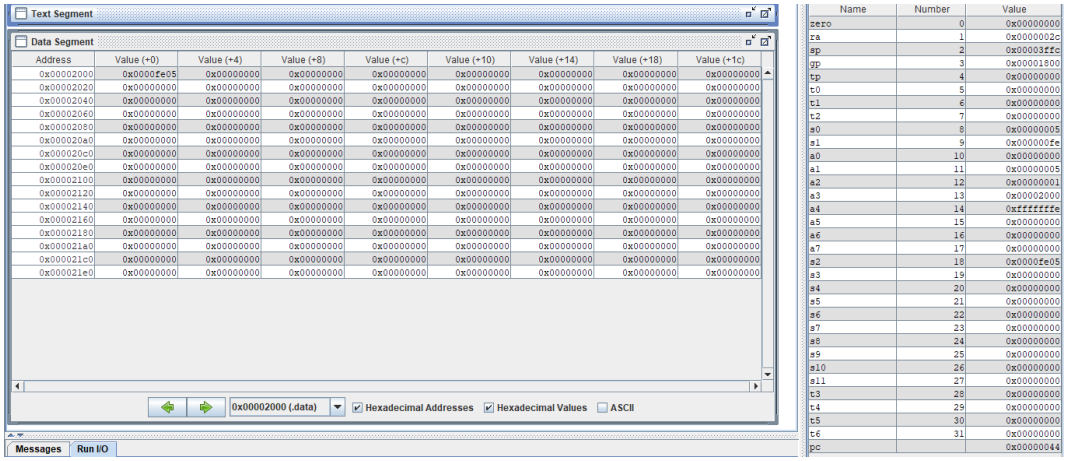
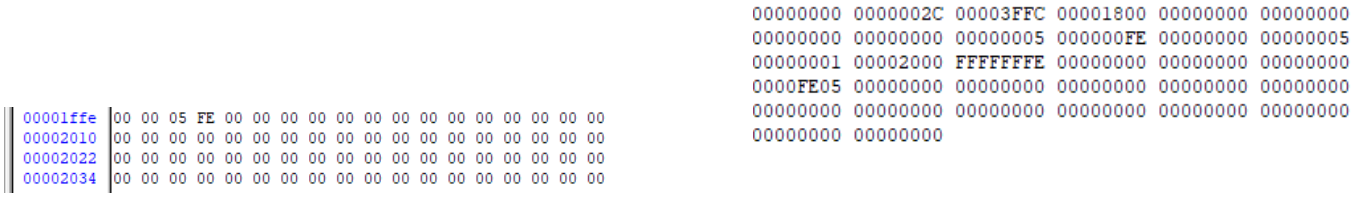


Figura 10: Tela do RARS para o código proposto



(a) Quadro de memória do ModelSim

(b) Quadro de registrador do ModelSim

Figura 11: A simulação confirma a igualdade entre o estado final no ModelSim e no RARS

4.5.2 JALR, Lógico-Aritméticas com/sem imediato

Code	Basic
0x0fe00513	addi x10,x0,0x000000fe
0x00054593	xori x11,x10,0
0x01400667	jalr x12,x0,20
0x00b58533	add x10,x11,x11
0x00800893	addi x17,x0,8
0x001a8a93	addi x21,x21,1
0x0158d4b3	srl x9,x17,x21
0x00400813	addi x16,x0,4
0x00081263	bne x16,x0,0x00000004
0x00000013	addi x0,x0,0

Figura 12: Código efetivo

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00003ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x000000fe
a1	11	0x000000fe
a2	12	0x0000000c
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000004
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000001
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x0000002c

Figura 13: Registradores no RARS

00000000	00000000	00000000	00003FFC	00001800
00000004	00000000	00000000	00000000	00000000
00000008	00000000	00000000	000000FE	000000FE
0000000c	0000000C	00000000	00000000	00000000
00000010	00000004	00000000	00000000	00000000
00000014	00000000	00000001	00000000	00000000
00000018	00000000	00000000	00000000	00000000
0000001c	00000000	00000000	00000000	00000000

Figura 14: Registradores no ModelSim, novamente confirmando o modelo desenvolvido

4.5.3 AUIPC

Code	Basic
0x00200813	addi x16,x0,2
0x00064817	auipc x16,0x00000064

Figura 15: Código efetivo

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00003ffc
gp	3	0x00001800
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00064004
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x0000000c

Figura 16: Registradores RARS

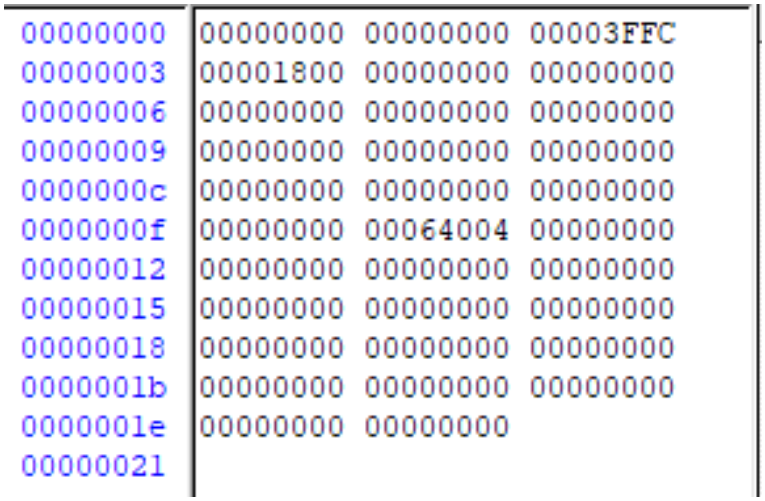


Figura 17: Registradores ModelSim, de encontro com o RARS

5 Conclusão

Em suma, a implementação do processador foi bem-sucedida. Foi possível rodar código gerado a partir do RARS contendo as instruções implementadas, embora o modelo apresente limitações pela memória de dados não poder ser carregada diretamente a partir do campo ".data".

Verificou-se que, embora otimizado do ponto de vista abstrato, o modelo desenvolvido pode complicar a implementação em Hardware, fato que deve-se atentar em trabalhos futuros.