Universidade Federal do Paraná

CI1210 - PROJETOS DIGITAIS E MICROPROCESSADORES BCC1

PROCESSADOR RISC-V SINGLE CYCLE

Aluno: **Ruibin Mei**
Professor: **Daniel Oliveira**

Curitiba, 2023

Sumário

1 INTRODUÇÃO

Nos últimos anos a arquitetura RISC-V vem sendo muito falado no cenário de design de processadores, sendo aberta, modular e eficiente. Entre muitas delas, o *single cycle* (monociclo), o que permite simplificar a execução de instruções, otimização do desempenho e a eficiência do processador.

A arquitetura RISC-V, desenvolvida com propósito de ser *open source*, ou seja, sendo aberta permitindo que qualquer pessoa possa utilizar, modificar e distribuir sem restrições. Diferenciando de muitas arquiteturas populares como x86 e ARM.

Além disso, RISC-V, é uma arquitetura RISC (*Reduced Instruction Set Computing*), que possui as caracteristicas de ser conjuntos de instruções reduzidas e simples, focando no desempenho otimizado e com mais eficiencia, além do seu acesso a memória por instruções *load/store*. Diferente de uma arquitetura CISC (*Complex Instruction Set Computing*), que possui conjuntos de instruções mais complexas e variados, podendo realizar tarefas mais sofisticadas e realizar várias operações em uma única instrução, o que indica ser uma arquitetura que precisa ser *multicycly* (vários ciclos de clock) e o seu acesso a memória é usado uma instrução de operação lógica ou aritmética. Algumas arquiteturas notáveis de RISC é o RISC-V, ARM e o MIPS, e CISC x86.

2 DESCRIÇÃO BREVE DO TRABALHO

No trabalho desenvolvido, utiliza-se um software chamado *"Digital"* para implementar uma microarquitetura monociclo seguindo a ISA RISC-V, com objetivo de testar se o circuito lógico foi elaborada de forma correta. Testando o algoritmo da sequência de *Fibonacci* começando com o primeiro termo como 1 e vai até o vigésimo termo 6765.

Um simulador de assembly *"RARS"* para implementar um programa em assembly para executar na microarquitetura elaborada.

3 IMPLEMENTAÇÃO DO CIRCUITO LÓGICO

A implementação do circuito lógico foi implementada seguindo um subconjunto da ISA RISC-V, mostrado na imagem abaixo (Figura 1), formato das instruções (Figura 2):

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm

Figura 1: Subconjunto da ISA RISC-V

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

Figura 2: Formato das Instruções

3.1 UNIDADE LÓGICA E ARITMÉTICA

A ALU (*arithmetic logic unit*), conhecida também como ULA (*Unidade lógica e aritmética*), foi a primeira parte feita neste trabalho. No nome ficará notável que fará operações lógicas e aritméticas do processador, seguindo um subconjunto da ISA RISC-V (Figura 1).

Analisando as instruções, algumas instruções são em comum entre algumas operações, como os do FMT (tipo da instrução), do *R-Type* e as do *I-Type*, na qual os de tipo R, fazem operações de ADD (adição), SUB (subtração), XOR (ou exclusivo), OR (ou), AND (e), SLL (*Shift Left Logical*, deslocamento para esquerda), SLT (*Set Less Than*, se rs1 é menor que rs2). E do tipo I, são operações do tipo *Immediate*, ou seja, imediato, então fará as mesma operações do tipo R, porém com valores constantes.

Assim, como outras instruções, *S-Type*, *B-Type*, *J-Type*, e as mais específicas como LW (*Load Word*), SW (*Store word*), JAL (*Jump And Link*) e JALR (*Jump And Link Register*), usam também algumas operações apresentados no tipo R, entretanto, foi feito na ULA apenas essas 7 operações, mostrado na Figura 3.

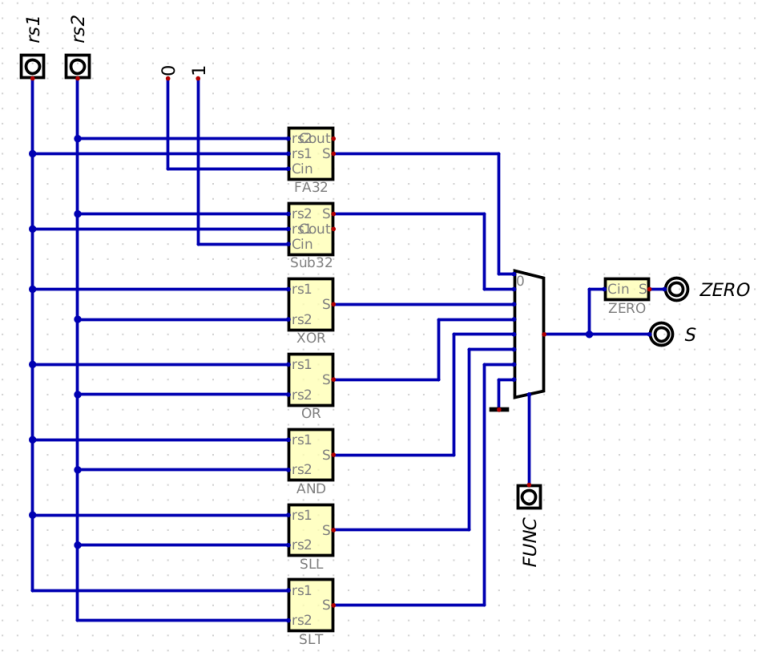
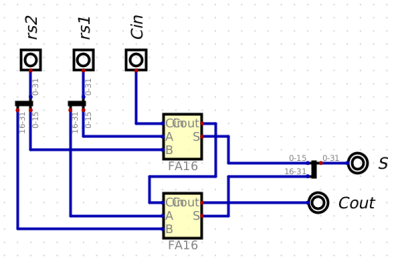


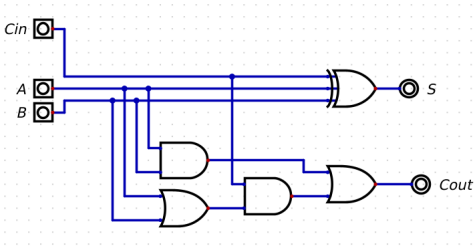
Figura 3: Circuito da ALU

3.1.1 Fa32

FA32, abreviado de *Full Adder 32 bits*, é um somador, conhecida como *Ripple Carry* de 32 bits com $Cin = 0$, mostrado na Figura 4a e na Figura 4b.



(a) Ripple Carry de 32 Bits



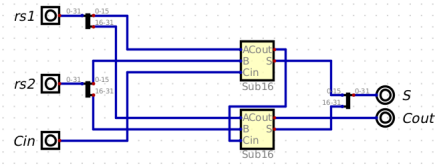
(b) Ripple Carry de 1 Bit

Figura 4: Circuitos dos Somadores

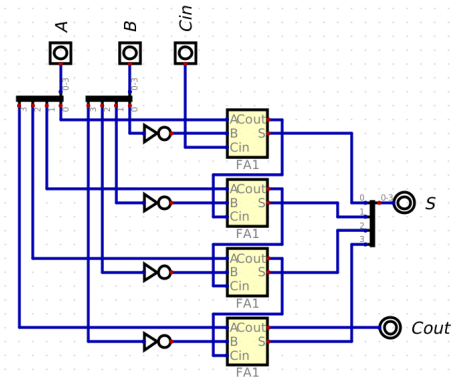
Nas imagens, o Somador de 32 bits ele utiliza dois somadores de 16 bits, e assim consequentemente, ou seja, ele é formado por varios somadores de 1 bit, como mostrado na imagem 4b.

3.1.2 Sub32

Sub32, abreviado de *Subtrator de 32 bits*, um somador porém com complemento de 2, para que faça a operação $rs1 + (-rs2)$, o que indica uma subtração, Figura 5a e Figura 5b.



(a) Subtrator de 32 Bits



(b) Subtrator de 4 Bits

Figura 5: Circuitos dos Subtratores

Nota que na Figura 5b, as entradas B, são todas negadas, ou seja, foi feito um complemento de 2 (invertendo todos os bits e somando 1 no *Cin*, Figura 3, tem a constante 1). E é usado os *Full Adders* (Figura 4b), para fazer a operação citada acima.

3.1.3 xor

XOR, a saída dele segue a tabela:

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

O circuito, Figura 6.

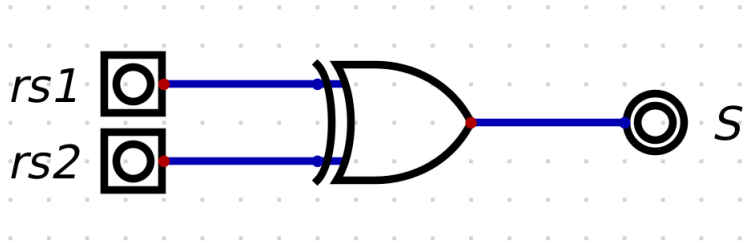


Figura 6: Circuito XOR

3.1.4 or

OR, a saída dele segue a tabela:

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

O circuito, Figura 7.

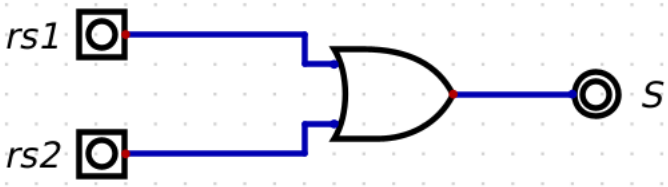


Figura 7: Circuito OR

3.1.5 and

AND, a saída dele segue a tabela:

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

O circuito, Figura 8

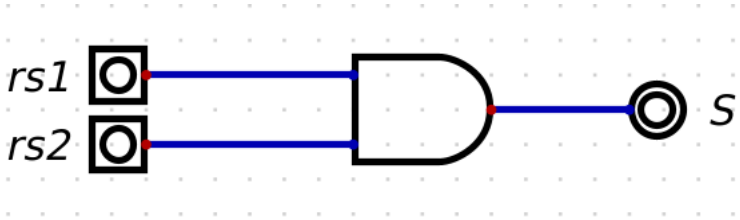


Figura 8: Circuito AND

3.1.6 Shitf less logical

SLL, é o *Shift Less Logical*, ou seja, vai fazer deslocamento para esquerda, exemplo: se a entrada rs1 é 00010 em binário, e a entrada rs2 é 00001, então o rs1 vai fazer um deslocamento de 1 bit (pois rs2 é igual a 1), resultando em 00100.

O circuito, Figura 9.

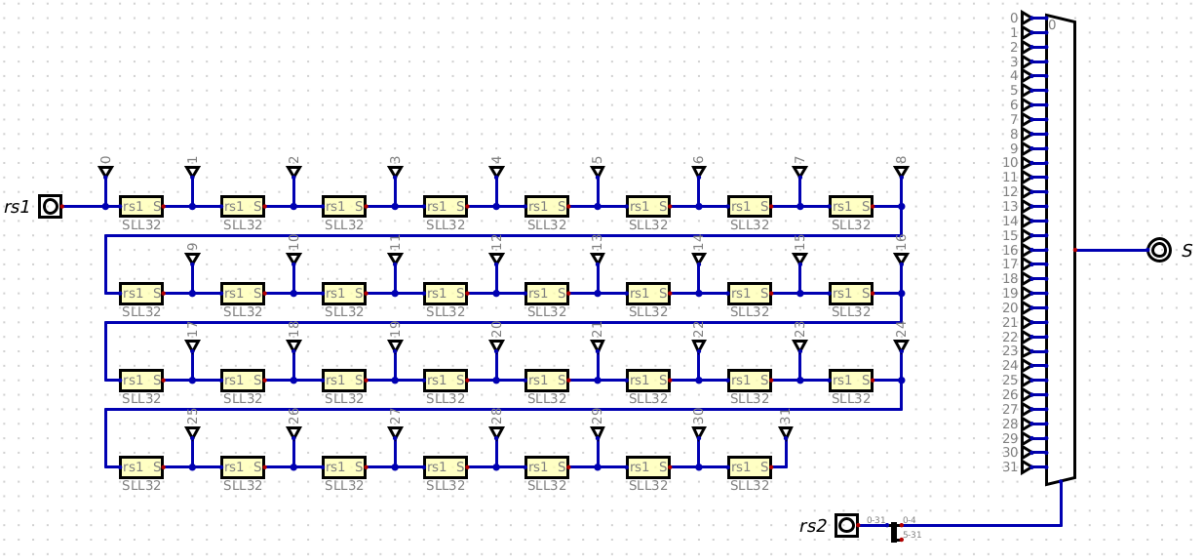
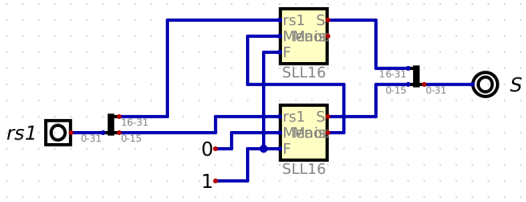
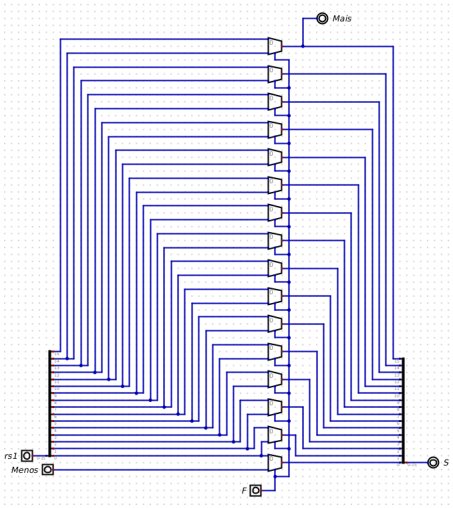


Figura 9: Circuito SLL

Para cada um dos SLL32, o circuito fica, Figura 10a e Figura 10b:



(a) SLL32



(b) SLL16

Figura 10: Circuito SLL32 e SLL16

3.1.7 Set less than

SLT, é o *Set Less Than*, na qual, faz uma comparação entre as duas entradas, rs1 e rs2, verificando a condição $rs1 < rs2$, se sim, a saída é 1 e se não, a saída é 0 (Figura 11).

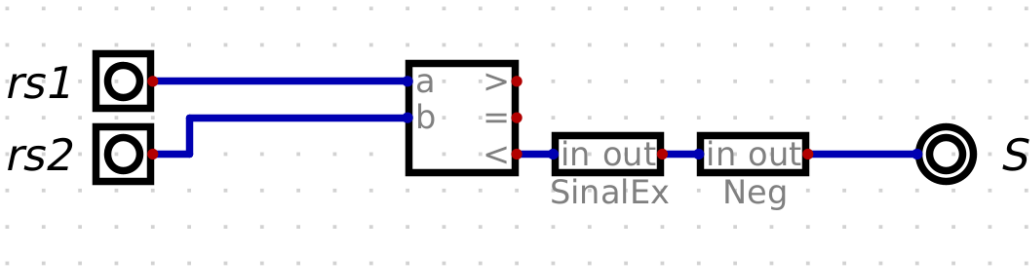


Figura 11: Circuito SLT

Nota que, foi usado o circuito já existente no software "*Digital*" um comparador, porém como a saída desse comparador é de 1 bit, e queremos na ULA a saída de 32 bits, então foi usado um extensor de sinais e foi colocado um complemento de 2, para que as saídas sem 1 ou 0 em 32 bits.

3.1.8 Zero

A função ZERO, na saída da ULA, facilita na hora da implementação do processador (Figura 12).

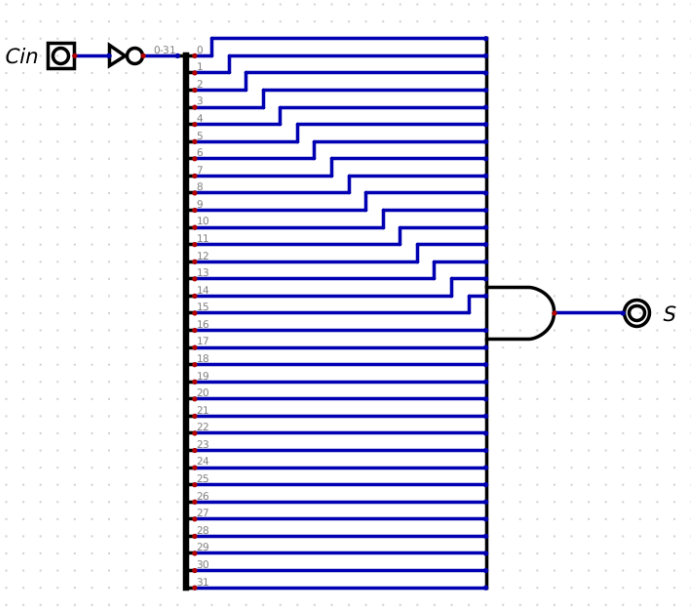


Figura 12: Circuito ZERO

No circuito da função ZERO, a entrada é negada e foi usada uma porta lógica AND, que permite uma saída de 1 bit. Ocorre neste circuito que se a entrada for de 32 bits de 0, vai virar 32 bits de 1, e a porta AND vai ter saída 1, ou seja, deu ZERO em todos os bits de entrada.

3.1.9 Func

Por fim, a função FUNC, que na verdade é a função/entrada, ou seja, ele será o seletor do MUX, para indicar qual operação será feito na ULA.

Os sinais de controle:

ALUControl	Function
000	ADD
001	SUB
010	XOR
011	OR
100	AND
101	SLL
110	SLT

3.2 EXTEND

O *EXTEND*, ele é um extensor de sinais, na qual, na hora de fazer um imediato quando necessário, ele será ativado pela instrução IMMSrc (*Immediate source*), sinal vinda da memória de controle (Figura 13).

O circuito segue o que é indicado pelo formato de instrução (Figura 2), ou seja, todos os bits ligado de forma exatamente igual o que é adequado para cada tipo de instrução.

Legenda:

- I_T sinal estendido para *I-Type*

- S_T sinal estendido para *S-Type*
- B_T sinal estendido para *B-Type*
- J_T sinal estendido para *J-Type*

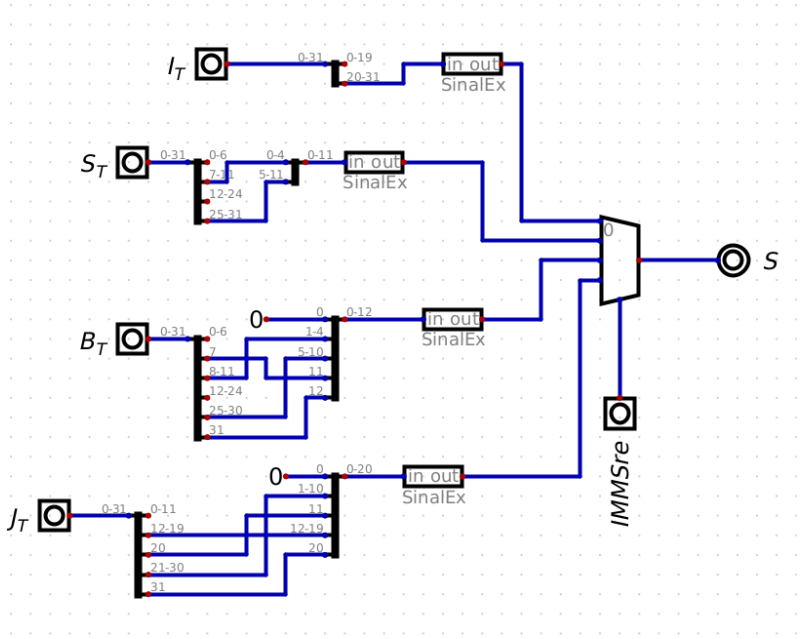


Figura 13: Circuito do imediato

3.3 MEMÓRIA DE INSTRUÇÕES

A memória de instruções é uma das partes mais importantes do processador, pois é ela que manda sinais para o processador, o que é feito em todo seu processamento. Ela trabalha junto com um PC ou IP, conhecido como *Program Counter* ou *Instruction Pointer*, sendo um registrador que mantém o endereço da próxima instrução a ser executada, em outras palavras, ela controla o fluxo do programa (Figura 14).

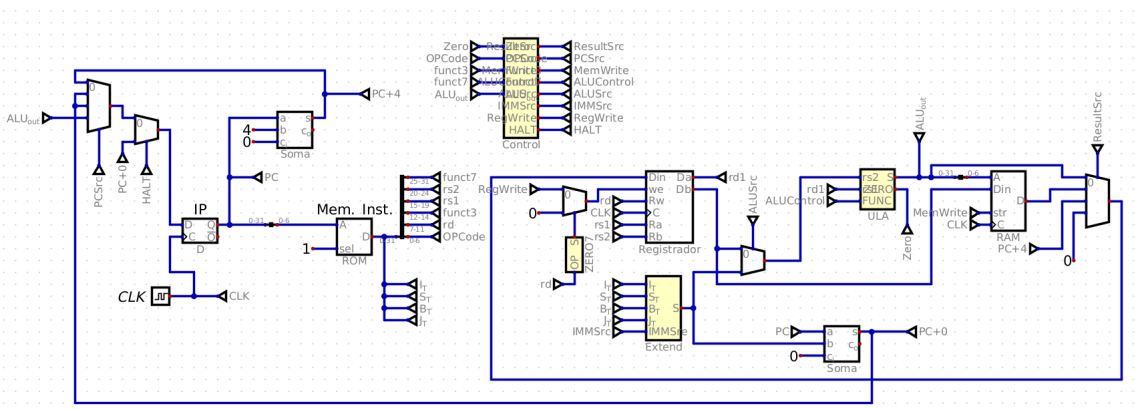


Figura 14: Circuito do RISC-V Single Cycle

Na implementação do todo o controle do fluxo, foi analisado o que acontece em cada uma das situações das instruções que ocorre. O PC elaborado, é um flip-flop tipo D junto

com uma ROM (*Read-Only Memory*, memória que pode ser apenas para leitura) e CLK (*Clock*) usada para atualizar o flip-flop e sincronização do circuito todo.

A ideia da implementação, foi iniciado primeiramente para que execute a próxima instrução, ou seja, como a memória no RISC-V é de byte a byte, então o $PC = PC + 4$, o que pode ser notado na parte esquerda superior do circuito, foi utilizado um somador e uma constante 4 para fazer $PC + 4$. Depois disso, como outras operações segundo a Figura 1 do tipo B, JAL e JALR, fazem $PC = PC + imm$, então foi preciso utilizar um MUX para receber o PC novo vindo ou do $PC + 4$, ou do $PC + imm$, na qual pode ser visto na parte inferior direita (um somador recebendo PC e o imediato).

O controle desse MUX, é o sinal do PCSrc (*PC source*), vindo da memória de controle. E além disso tem mais um MUX de duas entradas sendo selecionado pelo sinal *HALT*, essa operação foi elaborada para que o processador faça PC + 0, ou seja, ele não pode fazer absolutamente nada no processador, em outras palavras, ele termina o programa quando o sinal *HALT* é acionado.

Depois da memória de instruções é notável que possui vários distribuidores de sinais, como: *funtc7*, *rs2*, *rs1*, *funct3*, *rd*, *OPCode*, I_T, S_T,B_T, J_T, esses são os sinais que será necessário para memória de controle, banco de registradores e *EXTEND*.

Legenda:

- $funct7$ sinal de controle de operação
- $rs2$ endereço do registrador 2
- $rs1$ endereço do registrador 1
- $funct3$ sinal de controle de operação
- rd endereço do registrador destino
- $OPCode$ código da instrução

3.4 MEMÓRIA RAM

A RAM (*Random Access Memory*) é um tipo de memória volátil usada para armazenar dados temporariamente enquanto o sistema está em execução. Podendo ser vista na parte mais a direita do circuito todo na Figura 14. Ele é controlado pelo sinal MemWrite (*Memory Write*, escrita na memória), para que possa ser controlada se queremos ou não escrever nela.

Além disso, o sinal dele, da ULA, do PC + 4 e da constante 0, é controlado por um MUX, que recebe o sinal do *Result Source*, ou seja, em algumas operações como *JAL* e *JALR*, eles além de fazer $PC + imm$, o registrador destino recebe $rd = PC + 4$, então esse valor deve ser enviado para o Banco de registradores.

3.5 BANCO DE REGISTRADOR

O Banco de registrador (em cima do EXTEND na Figura 14) é uma peça importante para que podemos olhar nele alguns dados armazenados que é utilizado no processador quando algum programa é executado. Nele possui as entradas dos registradores, do dado, clock e um RegWrite (*Register Write*, um sinal de 1 bit vinda da memória de controle), para verificar se escreve ou não em algum registrador do branco de registradores.

Nele também tem um MUX que recebe o sinal do rd (registrador destino) para controle de não deixar o programa escrever no registrador x0 (zero). No RISC-V, o registrador x0, sempre será zero, não podendo ser sobrescrito, assim, esse MUX controla essa restrição.

3.6 MEMÓRIA DE CONTROLE

Com todas os componentes citadas acima, a memória de controle também é uma parte importante do processador, sem ela como no nome já indica, não terá controle de cada uma das componentes.

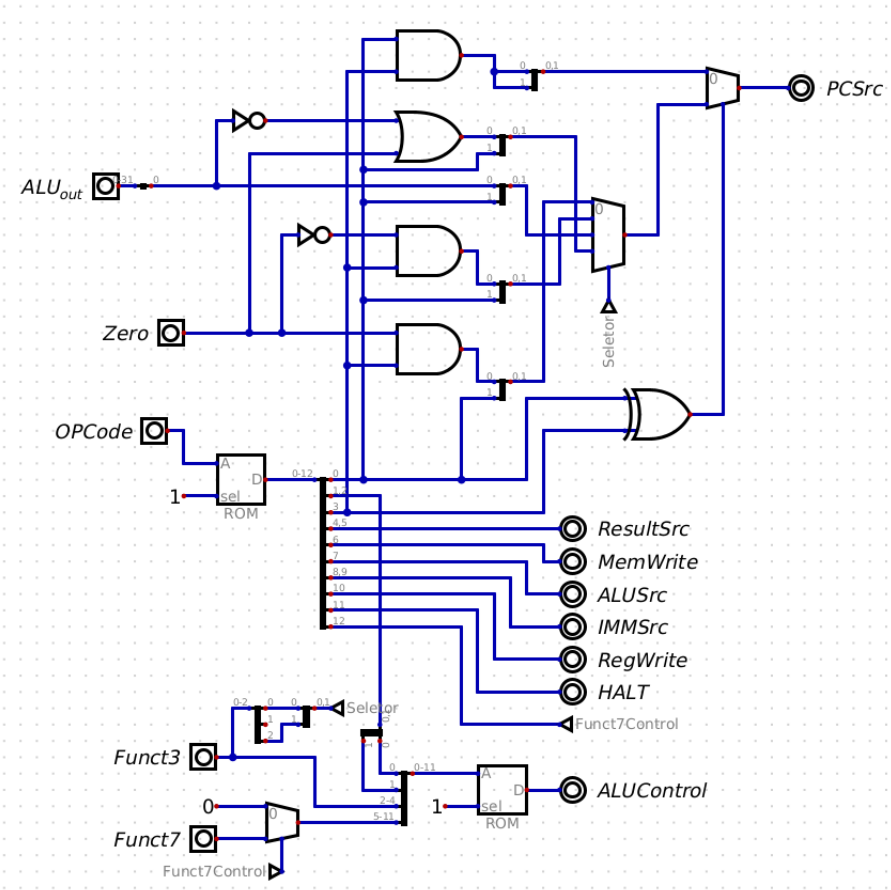


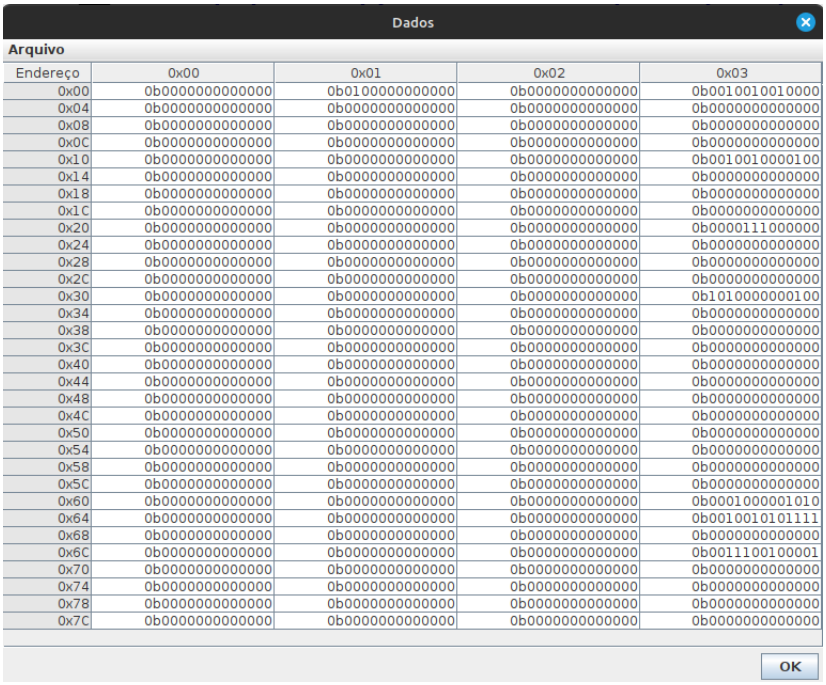
Figura 15: Memória de Controle

Na memória de controle ela recebe as entradas:

- ALU_{out} saída da ULA
- *Zero* sinal ZERO da ULA
- *OPCode* código da instrução
- *Funct*3 sinal de controle funct3
- *Funct*7 sinal de controle funct7

Primeiramente, foi usada duas ROMs para memória de controle, a primeira ROM (Figura 15) considerada como *MainControl*, controle principal, recebe o *OPCode* de 7

bits, que cada *OPCode* indica um endereço na ROM (Figura 16), e ele libera um sinal de 13 bits, com cada sinal de controle necessária.



Endereço	0x00	0x01	0x02	0x03
0x00	0b0000000000000	0b0100000000000	0b0000000000000	0b0010010010000
0x04	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x08	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x0C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x10	0b0000000000000	0b0000000000000	0b0000000000000	0b0010010000100
0x14	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x18	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x1C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x20	0b0000000000000	0b0000000000000	0b0000000000000	0b0000111000000
0x24	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x28	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x2C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x30	0b0000000000000	0b0000000000000	0b0000000000000	0b1010000000100
0x34	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x38	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x3C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x40	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x44	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x48	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x4C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x50	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x54	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x58	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x5C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x60	0b0000000000000	0b0000000000000	0b0000000000000	0b0001000001010
0x64	0b0000000000000	0b0000000000000	0b0000000000000	0b0010010101111
0x68	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x6C	0b0000000000000	0b0000000000000	0b0000000000000	0b0011100100001
0x70	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x74	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x78	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000
0x7C	0b0000000000000	0b0000000000000	0b0000000000000	0b0000000000000

Figura 16: ROM da MainControl

- 1 bit da instrução *Jump*
- 2 bits da *ALUOp* (*ALU Operation*)
- 1 bit da instrução *Branch*
- 1 bit de controle *ResultSrc*
- 1 bit da controle *MemWrite*
- 1 bit de controle *ALUsrc* (seleciona dado do registrador ou do imediato)
- 2 bits de controle *IMMSrc* (controle do MUX das operações de imediato do *EXTEND*)
- 1 bit de controle *RegWrite*
- 1 bit de controle *HALT*
- 1 bit de controle *Funct7Control* (controle do Funct7 na segunda ROM)

A segunda ROM (Figura 17), considerada como *ALUControl*, controla as saídas da *ALUControl*, para que seja feita operação correta na ULA. Ela recebe informação da *ALUOp* e dependendo das entradas de *Funct3* e *Funct7*, decide qual a saída para ULA. Informações em tabela:

ALUOp	Funct3	Funct7	Instrução	ALUControl
00	x	x	LW	000
	x	x	SW	000
01	000	x	BEQ	001
	001	x	BNE	001
	100	x	BLT	110
	101	x	BGE	110
10	000	0000000	ADD	000
	000	0100000	SUB	001
	100	0000000	XOR	010
	110	0000000	OR	011
	111	0000000	AND	100
	001	0000000	SLL	101
	010	0000000	SLT	110
11	000	x	JALR	000

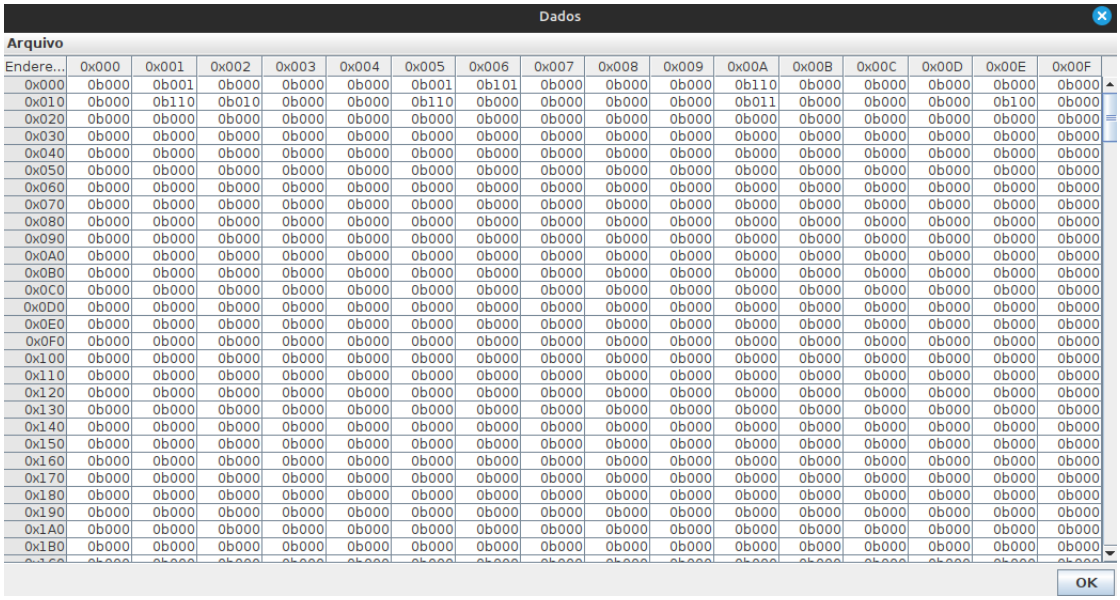


Figura 17: ROM da ALUControl

Nota que na entrada da *Funct7* tem um MUX e o controle dela é do sinal *Func7Control*, ou seja, como nas instruções do subconjunto da ISA RISC-V (Figura 1), *Funct7* controla apenas a operação *SUB*, então quem decide isso é a *Func7Control* vindo da *MainControl*.

Por fim, foi implementado as saídas do *PCSrc* (*PC source*), o que controla o MUX para decidir o valor do PC no próximo clock do processador. Para o controle das saídas das operações do *Branch*, foi usado um MUX recebendo o controle *Seletor* da *Funct3* do bit mais significativo e menos significativo, seguindo a tabela:

00	BEQ
01	BNE
10	BLT
11	BGE

A lógica usada para as saídas foi:

- BEQ $->$ sinal ZERO da ULA e sinal da operação Branch
- BNE $->$ sinal negado do ZERO da ULA e sinal da operação Branch
- BLT $->$ sinal do ALU_{out} da operação SLT
- BGE $->$ sinal negado do ALU_{out} da operação SLT ou ZERO da ULA

Nas quatros saídas os sinal para *PCSrc* será 01 (binário), e a saída para a operação JALR, é a porta lógica AND da parte superior, o que gera um sinal 11 para *PCsrc*, e quem controla esses sinais, é um XOR das instruções de Branch e Jump da tabela abaixa:

Instrução	OPCode	Funct7	HALT	RegWrite	IMMSrc
R-Type	0110011	1	0	1	xx
I-Type	0010011	0	0	1	00
S-Type	0100011	0	0	0	01
B-Type	1100011	0	0	0	10
J-Type	1101111	0	0	1	11
LW	0000011	0	0	1	00
JALR	1100111	0	0	1	00
HALT	0000001	0	1	0	00

ALUSrc	MemWrite	ResultSrc	Branch	ALUop	Jump
0	0	00	0	10	0
1	0	00	0	10	0
1	1	xx	0	00	0
0	0	xx	1	01	0
x	0	10	0	xx	0
1	0	01	0	00	0
1	0	10	1	11	1
0	0	0	0	00	0

4 ASSEMBLY

Assembly é uma linguagem de programação de baixo nível, ou seja, ela é específica para nível de arquitetura do computador, e o assembly utilizado neste trabalho, é específico da ISA RISC-V.
O algoritmo em linguagem C:

```
void fib(int * vet, int tam, int elem1, int elem2){
    int i;
    vet[0] = elem1;
    vet[1] = elem2;
    for(i=2; i<tam; ++i){
        vet[i] = vet[i-1] + vet[i-2];
    }
}
```

```c
int main(){
    int vet[20];
    fib(vet, 20, 1, 1);
}
```

O código traduzido para assembly RISC-V:

```
.text
   addi a0, gp, 0 # Iniciando vetor na posicao 0, com global pointer
   addi s0, zero, 20 # Registrador global recebe tamanho do vetor
   addi t0, zero, 1 # Registrador temporario recebe 1 elemento
   addi t1, zero, 1 # Registrafor temporario recebe 2 elemento
   jal ra, fib # Pula para funcao fib
fib:
   addi s1, zero, 2 # Registrador global recebe o indice 2 do laco
   sw t0, 0(a0) # Armazena no endereco 0 o 1 elemento, v[0]
   sw t1, 4(a0) # Armazena no endereco 4 o 2 elemento, v[1]
   addi a0, a0, 4 # O endereco agora esta no v[2]
fib_loop:
   add t2, t0, t1 # Faz uma adicao do v[i - 1] com o v[i - 2] elemento
   sw t2, 4(a0) # Guarda o valor da variavel t2 na posicao v[a0] + 4
   addi a0, a0, 4 # O endereco novo de a0  v[a0] + 4
   addi t0, t1, 0 # Guarda o elemento v[i - 2] em v[i - 1]
   addi t1, t2, 0 # Guarda o elemento t2 em v[i - 2]
   addi s1, s1, 1 # Faz i++
   blt s1, s0, fib_loop # Volta para fib_loop se indice e menor que tamanho 20
```

O código de assembly acima, foi testado usando o software *"RARS"*, os resultados da sequência de *Fibonacci* apresentado na Figura 18.

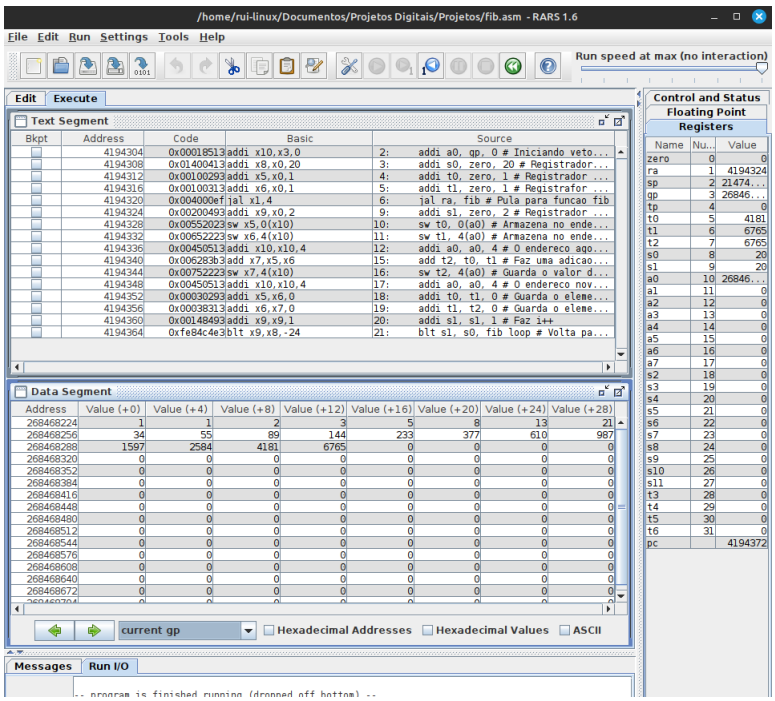


Figura 18: Resultados do algoritmo na RAM

5 RESULTADO

Com o circuito implementado, e o código em assembly feito e sendo testado no *"RARS"*, o último passo é passar as instruções seguindo o formato de instrução (Figura 2) para binário, ou hexadecimal e colocar na memória de instruções do circuito.

Hexadecimal	Binário	Assembly
1400413	00000001010000000000010000010011	addi s0, zero, 20
100293	00000000000010000000000001010010011	addi t0, zero, 1
100313	00000000000010000000000001100010011	addi t1, zero, 1
8000ef	00000000100000000000000011101111	jal ra, 8
1	00000000000000000000000000000001	HALT
200493	00000000000100000000010010010011	addi s1, zero, 2
552023	00000000010101010010000000100011	sw t0, 0(a0)
652223	00000000011001010010001000100011	sw t1, 4(a0)
450513	00000000010001010000010100010011	addi a0, a0, 4
6283b3	00000000011000101000001110110011	add t2, t0, t1
752223	00000000011101010010001000100011	sw t2, 4(a0)
450513	00000000010001010000010100010011	addi a0, a0, 4
30293	00000000000000011000001010010011	addi t0, t1, 0
38313	00000000000000011100001100010011	addi t1, t2, 0
148493	00000000000101001000010010010011	addi s1, s1, 1
fe84c4e3	11111110100001001100010011100011	blt s1, s0, -24
80e7	00000000000000001000000011100111	jalr x1, ra, 0

Na memória de instruções fica (Figura 19):

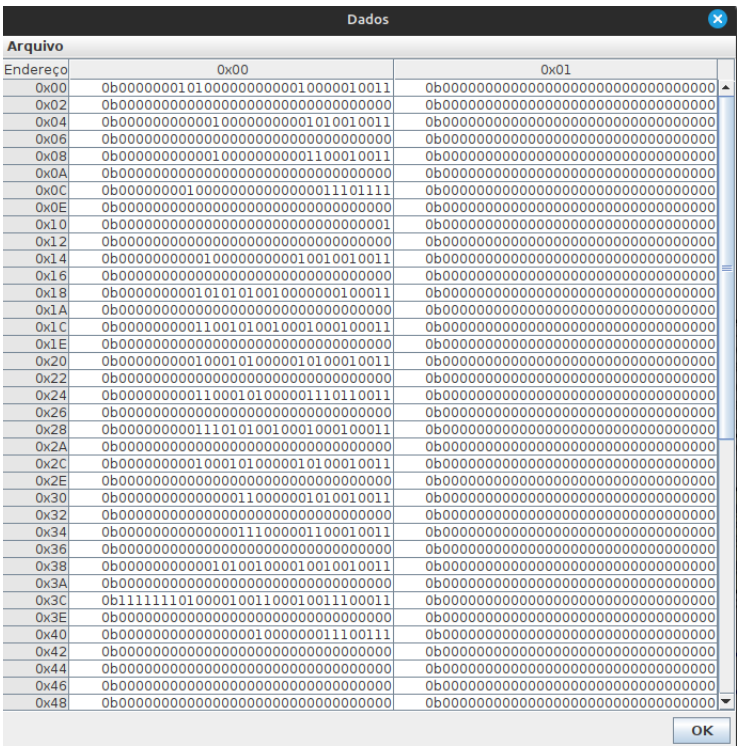


Figura 19: Memória de instruções

Os resultados obtidos ao testar o circuito (Figura 20 e Figura 21):

Registrador

End...	0x00	0x01	0x02	0x03
0x00	0x00000000	0x00000044	0x00000000	0x00000000
0x04	0x00000000	0x00001055	0x00001A6D	0x00001A6D
0x08	0x00000014	0x00000014	0x0000004C	0x00000000
0x0C	0x00000000	0x00000000	0x00000000	0x00000000
0x10	0x00000000	0x00000000	0x00000000	0x00000000
0x14	0x00000000	0x00000000	0x00000000	0x00000000
0x18	0x00000000	0x00000000	0x00000000	0x00000000
0x1C	0x00000000	0x00000000	0x00000000	0x00000000

Figura 20: Resultado do registrador

RAM

End...	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	0x00000001	0x00000000	0x00000000	0x00000000	0x00000001	0x00000000	0x00000000	0x00000000
0x08	0x00000002	0x00000000	0x00000000	0x00000000	0x00000003	0x00000000	0x00000000	0x00000000
0x10	0x00000005	0x00000000	0x00000000	0x00000000	0x00000008	0x00000000	0x00000000	0x00000000
0x18	0x0000000D	0x00000000	0x00000000	0x00000000	0x00000015	0x00000000	0x00000000	0x00000000
0x20	0x00000022	0x00000000	0x00000000	0x00000000	0x00000037	0x00000000	0x00000000	0x00000000
0x28	0x00000059	0x00000000	0x00000000	0x00000000	0x00000090	0x00000000	0x00000000	0x00000000
0x30	0x000000E9	0x00000000	0x00000000	0x00000000	0x00000179	0x00000000	0x00000000	0x00000000
0x38	0x00000262	0x00000000	0x00000000	0x00000000	0x000003DB	0x00000000	0x00000000	0x00000000
0x40	0x0000063D	0x00000000	0x00000000	0x00000000	0x00000A18	0x00000000	0x00000000	0x00000000
0x48	0x00001055	0x00000000	0x00000000	0x00000000	0x00001A6D	0x00000000	0x00000000	0x00000000
0x50	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x58	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x60	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x68	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x70	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x78	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figura 21: Resultado da RAM

Referências

[1] PATTERSON, David e WATERMAN, Andrew. *Guia prático RISC-V: Atlas de uma Arquitetura Aberta*. 1ª edição, 1.0.0, março de 29 de 2019.

[2] HEXSEL, Roberto A. *Sistemas Digitais e Microprocessadores*. ISBN 978-8573353068. Editora da UFPR, 2012.

[3] ZANATA, Marco. Curso de Projetos Digitais e Microprocessadores. 2023. Disponível em `https://www.youtube.com/playlist?list=PL_9px37PNj6oIJcyAxUSZ-Z8yLA2XS4gB`. Acesso em 28 de outubro de 2023.

[4] HARRIS, David e HARRIS, Sarah. RISC-V David Harris and Sarah Harris Microarchitecture. 2022. Disponível em `https://www.youtube.com/playlist?list=PLhA3DoZr6boVQy9Pz-aPZLH-rA6DvUidB`. Acesso em 26 de outubro de 2023.

[5] Tecnoblog. "Qual é a diferença entre arquitetura RISC e CISC em processadores."Disponível em `https://tecnoblog.net/responde/qual-e-a-diferenca-entre-arquitetura-risc-e-cisc-processador/`. Acesso em 15 de novembro de 2023.

[6] OLIVEIRA, Danilo. Editado por IGNACIO, Lima Ignacio. O que é a arquitetura RISC-V de processadores?. 2023. Disponível em `https://olhardigital.com.br/2023/06/20/reviews/o-que-e-a-arquitetura-risc-v-de-processadores/`. Acesso em 15 de novembro de 2023.