Relatório do Trabalho de ISA e Microarquitetura

1. Especificação da ISA

1.1 Introdução

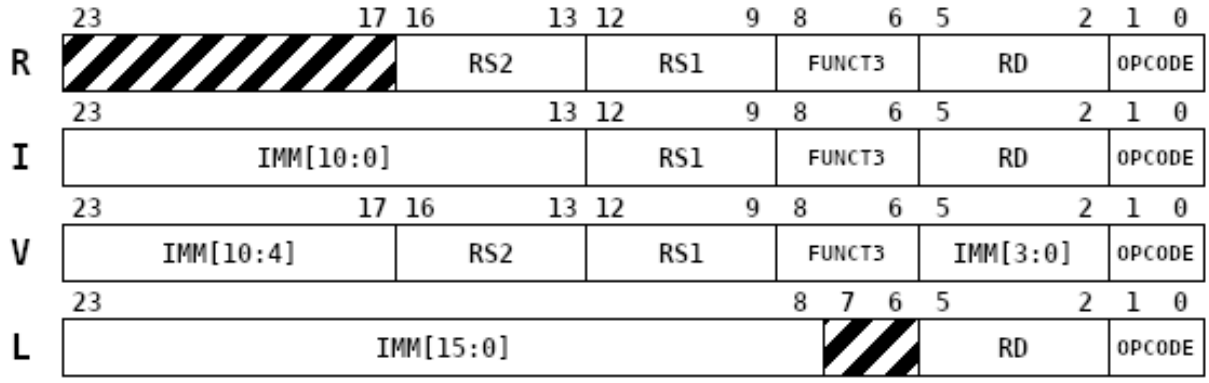
O conjunto de instruções desenvolvido neste trabalho foi fortemente inspirado nas ISAs RV32I e RV32M da arquitetura RISC-V, adaptadas para instruções de 24 bits. Devido à limitação de tamanho para a instrução, foram incluídas apenas operações essenciais. Ao todo, o conjunto de instruções contém 23 operações.

O objetivo foi manter a arquitetura o mais simples possível, sem comprometer a praticidade de uso. O conjunto de instruções definido cobre operações aritméticas, lógicas, controle de fluxo e acesso à memória, proporcionando suporte completo às estruturas básicas de programação e chamadas de função.

1.2 Formatos de instrução

A ISA proposta possui quatro formatos de instrução, cada um identificado unicamente pelo `opcode`:

- **Formato R:** Utilizado para operações lógicas e aritméticas entre registradores. Contém dois registradores fonte (`rs1` e `rs2`) e um registrador destino (`rd`).
- **Formato I:** Usado para operações que envolvem um registrador fonte (`rs1`), um valor imediato (`imm`) e um registrador destino (`rd`). Além de operações lógicas e aritméticas, é usado para operações de salto e acesso à memória.
- **Formato V:** Usado para operações que envolvem dois registradores fonte (`rs1, rs2`) e um valor imediato (`imm`), como instruções de *branch* e *store*.
- **Formato L:** Exclusivo para a operação `li (load immediate)`, usada para carregar um valor imediato de 16 bits (`imm`) em algum registrador (`rd`).



Todos os formatos incluem 2 bits de `opcode`, suficientes para determinar o tipo da instrução. Além disso, os formatos R, I e V possuem 3 bits adicionais (`funct3`), permitindo 8 instruções únicas para cada formato. O formato L não utiliza `funct3`, liberando espaço para um imediato maior.

Os formatos R e L possuem bits excedentes que poderiam ser usados para expandir o conjunto de instruções, mas optou-se por mantê-los inutilizados para simplificar a implementação.

1.3 Registradores

O banco de registradores inclui 16 registradores de propósito geral, cada um com 32 bits, organizados conforme a seguinte convenção de uso:

- `r0 (zero)`: constante zero, imutável
- `r1 (rad)`: endereço de retorno (return address)

- `r2 (rbp)`: ponteiro para a base da pilha (base pointer)
- `r3 (rsp)`: ponteiro para o topo da pilha (stack pointer)
- `r4..r7 (rt0..rt3])`: registradores temporários
- `r8..r11 (rs0..rs3)`: registradores salvos
- `r12 (ra0)`: valor de retorno / primeiro argumento de função
- `r13..r15 (ra1..ra3)`: argumentos de função

Há também um registrador especial de 8 bits reservado para o controle do fluxo de execução: o registrador `ip` (`instruction pointer`). Ele é incrementado automaticamente após a execução de cada instrução, exceto em casos de desvios e saltos, onde o valor é alterado diretamente.

1.4 Conjunto de instruções

1.4.1 Instruções lógicas e aritméticas

Formato R:
- `add`: soma.
- `sub`: subtração.
- `mul`: multiplicação.
- `div`: divisão inteira.
- `mod`: resto da divisão inteira.
- `and`, `or`, `xor`: operações lógicas.

Formato I:
- `addi`: soma com valor imediato.
- `modi`: resto da divisão inteira com valor imediato.
- `slli`: *shift* lógico para a esquerda com valor imediato.
- `and`, `or`, `xor`: operações lógicas com valor imediato.

1.4.2 Instruções de acesso à memória

Formato I:
- `lw`: Carrega uma palavra de 32 bits da memória para o registrador.

Formato V:
- `sw`: Armazena uma palavra de 32 bits na memória a partir de um registrador.

As operações de acesso à memória utilizam a notação '`op rx, imm(ry)`', onde `ry` é o registrador que contém o endereço base e `imm` é um valor de deslocamento.

1.4.3 Instruções de salto / desvio de fluxo

Formato I:
- `jalr`: Salto incondicional para o endereço em `rs1` deslocado de `imm`; `rd` recebe o endereço de retorno.

Formato V:
- `beq`, `bne`: Desvio condicional para igualdade e diferença. Se a condição for verdadeira, salta para `ip` + `imm`.
- `bge`, `blt`: Desvio condicional para maior ou igual e menor.
- `halt`: para a execução do programa, impedindo `ip` de ser incrementado.

É importante notar que as instruções de desvio condicional (branch) realizam saltos relativos à posição atual no programa, enquanto as instruções de salto (jump) utilizam endereçamento absoluto, desconsiderando a posição atual.

1.4.4 Instruções de carregamento

Formato L:

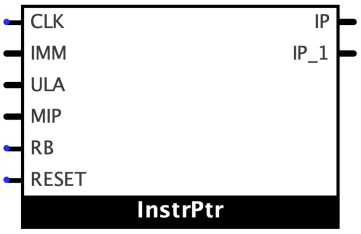
- `li`: Carrega um imediato de 16 bits (`imm`) na parte baixa do registrador `rd`.

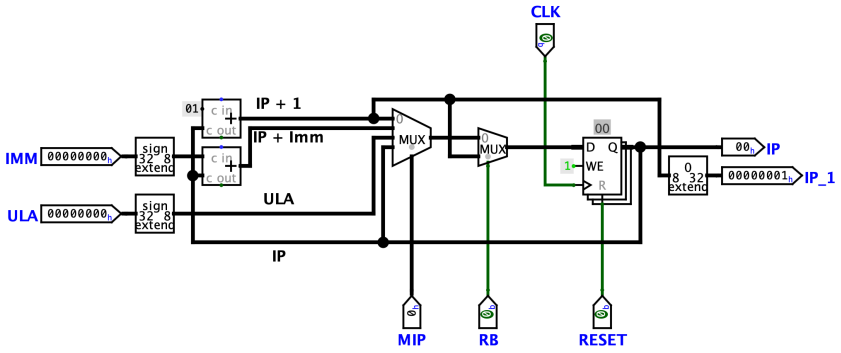
2. Implementação da ISA

A implementação da ISA foi estruturada em módulos independentes, posteriormente integrados em um circuito principal para alcançar a funcionalidade desejada.

2.1 InstrPtr

Este circuito controla o registrador `ip` (`instruction pointer`), de 8 bits. Com base nas entradas, decide qual será o valor de `ip` no próximo ciclo de clock.





Entradas:

- CLK (1b): Entrada para o clock;
- IMM (32b): O valor do imediato para a instrução atual;
- ULA (32b): O resultado da operação calculada pela unidade de lógica e aritmética;
- MIP (2b): Valor de seleção calculado pela unidade de controle para o multiplexador que escolhe entre `ip + 1`, `ip + imm`, `ULA` e `ip`;
- RB (1b): Valor de seleção para o segundo multiplexador, calculado pelo circuito de branch;
- RESET (1b): Caso RESET seja 1, o valor de `ip` é zerado.

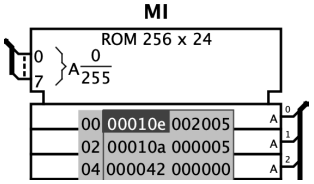
Saídas:

- IP (8b): O valor atual do registrador `ip`;
- IP_1 (32b): O valor atual do registrador `ip` incrementado em 1.

MIP e RB são usados em conjunto para decidir se a instrução atual irá incrementar o registrador `ip` ou se irá fazer um desvio/salto. A saída IP_1 é estendida para 32 bits, pois é usada pela instrução `jalr` (que guarda o endereço de retorno no registrador de destino, de 32 bits).

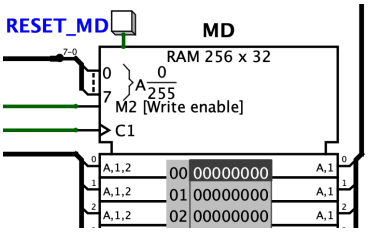
2.2 MI (Memória de Instruções)

Uma memória ROM com 256 endereços de 24 bits.
Cada endereço guarda uma instrução.



2.3 MD (Memória de Dados)

Uma memória RAM com 256 endereços de 32 bits.

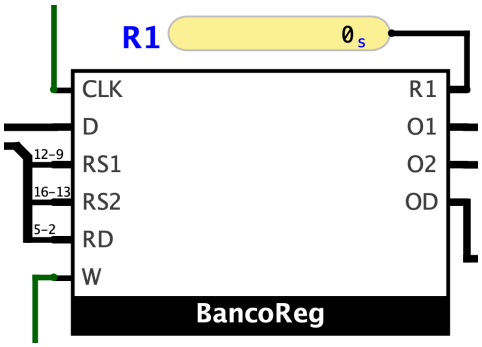


2.4 Imm

Um circuito que calcula o valor do imediato com base na instrução atual, considerando os formatos definidos. O resultado é o valor do imediato estendido para 32 bits.

2.5 BancoReg

Um banco de registradores com 16 registradores de 32 bits.



Entradas:

- CLK (1b): Entrada para o clock;
- D (32b): Valor de 32 bits que para ser escrito em RD;
- RS1 (4b): Código do registrador `rs1`;
- RS2 (4b): Código do registrador `rs2`;
- RD (4b): Código do registrador `rd`;
- W (1b): *write enable* (1 habilita a escrita, 0 desabilita).

Saídas:

- R1 (32b): O valor atual do registrador `r1`;
- O1 (32b): O valor atual do registrador `rs1`;
- O2 (32b): O valor atual do registrador `rs2`;
- OD (32b): O valor atual do registrador `rd`.

A saída R1 foi implementada com o único propósito de facilitar o debugging, pois permite visualizar o valor de um registrador sem ter que executar instruções de acesso à memória.

A saída OD foi implementada para possibilitar que a operação `li` (`load immediate`) altere a parte baixa [15:0] do registrador `rd` sem alterar sua parte alta [31:16].

2.6 ULA (Unidade de Lógica e Aritmética)

Um circuito que, dados um código de operação e dois operandos de 32 bits, calcula o resultado da operação lógica ou aritmética representada por OPR.

Entradas:

- OPR (4b): Código de operação;
- OP1 (32b): Primeiro operando;
- OP2 (32b): Segundo operando.

Saídas:

- RES (32b): O resultado da operação.

OPR	EF	OPR	EF
0000	add	1000	add
0001	sub	1001	add
0010	mul	1010	add
0011	div	1011	sll
0100	mod	1100	mod
0101	and	1101	and
0110	or	1110	or
0111	xor	1111	xor

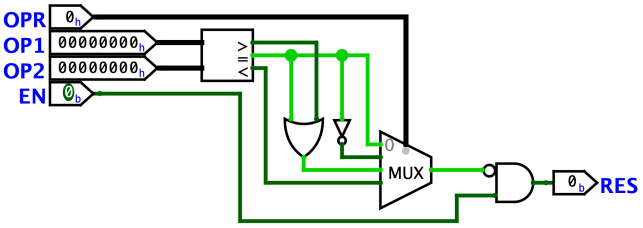
Existem operações com mais de um código associado. Isso foi feito para facilitar a implementação da unidade de controle. Como havia a necessidade de implementar 9 operações na ULA, seriam necessários 4 bits para representar uma operação. Os códigos de operação foram alocados de modo a coincidirem com o valor formado pelo primeiro bit do `opcode` seguido do `funct3`.

2.7 Load

Um circuito usado exclusivamente pela instrução `li` (`load immediate`). A saída RES é o valor de 32 bits formado por BASE[31:16] e IMM[15:0].

2.8 Branch

Circuito usado pelas instruções de branch. Dados um código de operação e dois operandos de 32 bits, calcula o valor para a entrada RB do circuito InstrPtr.



Entradas:

- OPR (2b): Código de operação;
- OP1 (32b): Primeiro operando;
- OP2 (32b): Segundo operando;
- EN (1b): Enable.

Saídas:

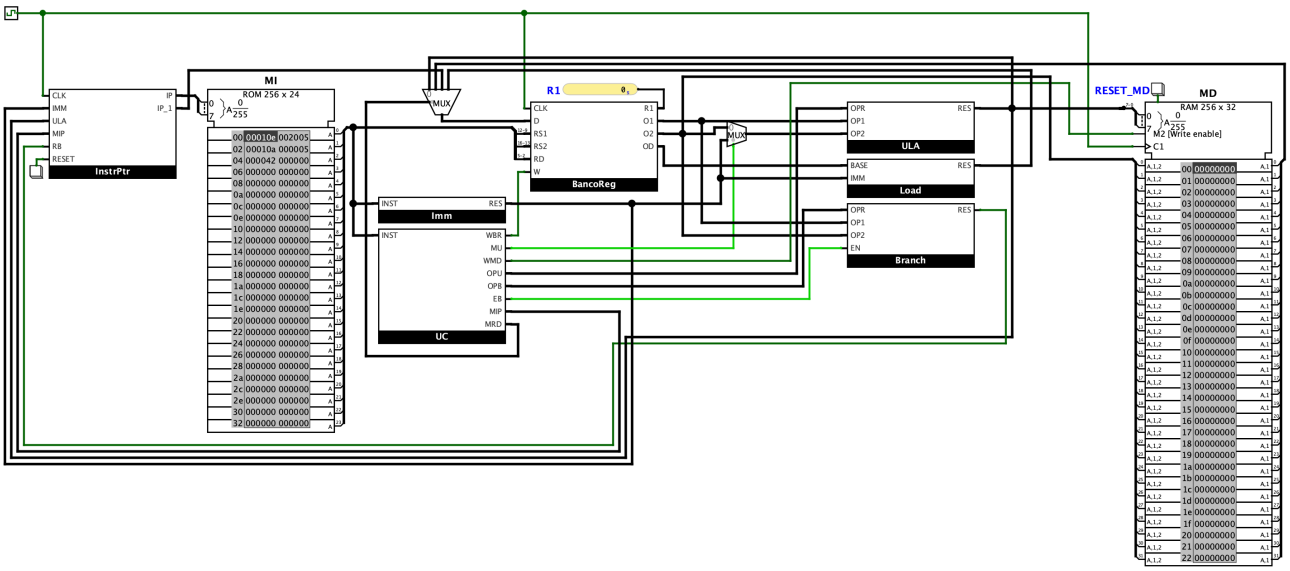
- RES (1b): O valor de RB para InstrPtr.

É importante notar que o valor RES desse circuito não é uma representação direta do resultado da comparação especificada em OPR, mas o valor de controle adequado para o segundo multiplexador de InstrPtr.

OPR	EF	EN	BRANCH?	RES
00	==	0	0	0
01	!=	0	1	0
10	>=	1	0	1
11	<	1	1	0

2.9 Circuito Principal

O circuito principal foi construído com os módulos especificados acima, além da UC (Unidade de Controle) e dois multiplexadores: MU (Mux da ULA) e MRD (Mux do Registrador Destino).

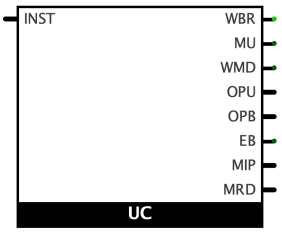


MU é usado para selecionar entre RS2 (0) e IMM (1) para o segundo operando da ULA (OP2).

MRD é usado para selecionar entre o resultado da ULA (00), MD (01), IP+1 (10) e o resultado do circuito de Load (11) para o valor que será escrito em RD.

2.10 UC (Unidade de Controle)

A unidade de controle é o circuito que gerencia os sinais de controle de todos os outros circuitos com base na instrução atual.



Entradas:

- INST (24b): A instrução atual.

Saídas:

- WBR (1b): Sinal de write para o banco de registradores;

- MU (1b): Sinal de controle para o Mux da ULA;
- WMD (1b): Sinal de write para a memória de dados;
- OPU (4b): Código de operação da ULA;
- OPB (2b): Código de operação do circuito de Branch;
- EB (1b): Sinal de enable para o circuito de Branch;
- MIP (2b): Sinal de controle para o Mux do IP (entrada MIP de InstrPtr);
- MRD (2b): Sinal de controle para o Mux do Registrador Destino.

FMT	INST	OPC	FN3	WBR	WMD	MU	OPU	OPB	EB	MIP	MRD
R	add	00	000	1	0	0	0000	XX	0	00	00
	sub	00	001	1	0	0	0001	XX	0	00	00
	mul	00	010	1	0	0	0010	XX	0	00	00
	div	00	011	1	0	0	0011	XX	0	00	00
	mod	00	100	1	0	0	0100	XX	0	00	00
	and	00	101	1	0	0	0101	XX	0	00	00
	or	00	110	1	0	0	0110	XX	0	00	00
	xor	00	111	1	0	0	0111	XX	0	00	00
I	addi	01	000	1	0	1	1000	XX	0	00	00
	lw	01	001	1	0	1	1001	XX	0	00	01
	jalr	01	010	1	0	1	1010	XX	0	10	10
	slli	01	011	1	0	1	1011	XX	0	00	00
	modi	01	100	1	0	1	1100	XX	0	00	00
	andi	01	101	1	0	1	1101	XX	0	00	00
	ori	01	110	1	0	1	1110	XX	0	00	00
	xori	01	111	1	0	1	1111	XX	0	00	00
V	sw	10	000	0	1	1	0000	XX	0	00	XX
	halt	10	001	0	0	1	XXXX	XX	0	11	XX
	beq	10	100	0	0	1	XXXX	00	1	01	XX
	bne	10	101	0	0	1	XXXX	01	1	01	XX
	bge	10	110	0	0	1	XXXX	10	1	01	XX
	blt	10	111	0	0	1	XXXX	11	1	01	XX
L	li	11	––	1	0	X	XXXX	XX	0	00	11
FMT	INST	OPC	FN3	WBR	WMD	MU	OPU	OPB	EB	MIP	MRD

A implementação foi feita por meio de circuitos combinacionais.

3. Tradução de um código em assembly na ISA definida

3.1 Script montador

Para traduzir o assembly para linguagem de máquina, foi utilizado o assembler customizável customasm com o seguinte conjunto de regras:

```
#subruledef register
{
    r0  => 0x0
    r1  => 0x1
    r2  => 0x2
    r3  => 0x3
    r4  => 0x4
    r5  => 0x5
    r6  => 0x6
    r7  => 0x7
    r8  => 0x8
    r9  => 0x9
    r10 => 0xA
    r11 => 0xB
    r12 => 0xC
    r13 => 0xD
    r14 => 0xE
    r15 => 0xF

    zero => 0x0
    rad  => 0x1
    rbp  => 0x2
    rsp  => 0x3
    rt0  => 0x4
    rt1  => 0x5
    rt2  => 0x6
    rt3  => 0x7
    rs0  => 0x8
    rs1  => 0x9
    rs2  => 0xA
    rs3  => 0xB
    ra0  => 0xC
    ra1  => 0xD
    ra2  => 0xE
    ra3  => 0xF
}

#ruledef
{
    add {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b00
    sub {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b001 @ rd @ 0b00
    mul {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b010 @ rd @ 0b00
    div {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b011 @ rd @ 0b00
    mod {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b100 @ rd @ 0b00
    and {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b101 @ rd @ 0b00
    or  {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b110 @ rd @ 0b00
    xor {rd: register}, {rs1: register}, {rs2: register} => 0b0000000 @ rs2 @ rs1 @ 0b111 @ rd @ 0b00

    addi {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b000 @ rd @ 0b01
    lw   {rd: register}, {imm: s11}({rs1: register}) => imm @ rs1 @ 0b001 @ rd @ 0b01
    jalr {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b010 @ rd @ 0b01
    slli {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b011 @ rd @ 0b01
    modi {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b100 @ rd @ 0b01
    andi {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b101 @ rd @ 0b01
    ori  {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b110 @ rd @ 0b01
    xori {rd: register}, {rs1: register}, {imm: s11} => imm @ rs1 @ 0b111 @ rd @ 0b01

    sw   {rs2: register}, {imm: s11}({rs1: register}) => imm[10:4] @ rs2 @ rs1 @ 0b000 @ imm[3:0] @ 0b10
    halt                                              => 0b0000000000000000 @ 0b001 @ 0b0000 @ 0b10
    beq  {rs1: register}, {rs2: register}, {imm: s11} => imm[10:4] @ rs2 @ rs1 @ 0b100 @ imm[3:0] @ 0b10
    bne  {rs1: register}, {rs2: register}, {imm: s11} => imm[10:4] @ rs2 @ rs1 @ 0b101 @ imm[3:0] @ 0b10
    bge  {rs1: register}, {rs2: register}, {imm: s11} => imm[10:4] @ rs2 @ rs1 @ 0b110 @ imm[3:0] @ 0b10
    blt  {rs1: register}, {rs2: register}, {imm: s11} => imm[10:4] @ rs2 @ rs1 @ 0b111 @ imm[3:0] @ 0b10

    li   {rd: register}, {imm: i16} => imm @ 0b00 @ rd @ 0b11
}
```

3.2 Código de teste

Este código testa todas as instruções da ISA. A coluna `addr` contém o endereço da memória de instruções, `binary` contém o código de máquina e `asm` contém o assembly. O código foi comentado com o comportamento esperado.

```
addr |          binary            | asm

  00 |                            | _start:
  00 | 00000010 01100000 00000101 | addi r1, r0, 19    ; r1 = 19
  01 | 00000000 10000000 00001001 | addi r2, r0, 4     ; r2 = 4
  02 | 00000000 01000010 00000100 | add  r1, r1, r2    ; r1 = 23
  03 | 00000000 00100100 01000100 | sub  r1, r2, r1    ; r1 = -19
  04 | 00000000 01000010 11000100 | div  r1, r1, r2    ; r1 = -4
  05 | 00000000 01000011 00000100 | mod  r1, r1, r2    ; r1 = 0
  06 | 00000000 01100000 00000101 | addi r1, r0, 3     ; r1 = 3
  07 | 00000000 01000011 01000100 | and  r1, r1, r2    ; r1 = 0
  08 | 00000000 01000011 10000100 | or   r1, r1, r2    ; r1 = 4
  09 | 00000000 01000011 11000100 | xor  r1, r1, r2    ; r1 = 0
  0a | 00000010 01100000 00000101 | addi r1, r0, 19    ; r1 = 19
  0b | 00000001 01000011 00000101 | modi r1, r1, 10    ; r1 = 9
  0c | 00000001 00000011 01000101 | andi r1, r1, 8     ; r1 = 8
  0d | 00000000 01100011 10000101 | ori  r1, r1, 3     ; r1 = 11
  0e | 00000001 11100011 11000101 | xori r1, r1, 15    ; r1 = 4
  0f | 11011110 10101101 00000111 | li   r1, 0xdead    ; r1 = 0xdead
  10 | 00000010 00000010 11000101 | slli r1, r1, 16    ; r1 = 0xdead0000
  11 | 10111110 11101111 00000111 | li   r1, 0xbeef    ; r1 = 0xdeadbeef
  12 | 00000000 00100000 00000010 | sw   r1, 0(r0)     ; MD[0] = 0xdeadbeef
  13 | 00000000 00000000 00000101 | addi r1, r0, 0     ; r1 = 0
  14 | 00000000 00000000 01000101 | lw   r1, 0(r0)     ; r1 = 0xdeadbeef
  15 | 11001010 11111110 00000111 | li   r1, 0xcafe    ; r1 = 0xdeadcafe
  16 | 00000010 00000010 11000101 | slli r1, r1, 16    ; r1 = 0xcafe0000
  17 | 00000000 00100000 00000110 | sw   r1, 1(r0)     ; MD[1] = 0xcafe0000
  18 | 00000000 00100000 00000101 | addi r1, r0, 1     ; r1 = 1
  19 | 00000000 01000000 00001001 | addi r2, r0, 2     ; r2 = 2
  1a | 00000000 00100011 00001010 | beq  r1, r1, A     ; salta para A
  1b | 11111111 11100000 00000101 | addi r1, r0, -1
  1b |                            | A:
  1c | 10000000 01000011 00000010 | beq  r1, r2, -1024 ; nao salta
  1d | 00000000 01000011 01001010 | bne  r1, r2, B     ; salta para B
  1e | 11111111 11100000 00000101 | addi r1, r0, -1
  1f |                            | B:
  1f | 10000000 00100011 01000010 | bne  r1, r1, -1024 ; nao salta
  20 | 00000000 00100101 10001010 | bge  r2, r1, C     ; salta para C
  21 | 11111111 11100000 00000101 | addi r1, r0, -1
  22 |                            | C:
  22 | 10000000 01000011 10000010 | bge  r1, r2, -1024 ; nao salta
  23 | 00000000 01000011 11001010 | blt  r1, r2, D     ; salta para D
  24 | 11111111 11100000 00000101 | addi r1, r0, -1
  25 |                            | D:
  25 | 10000000 00100101 11000010 | blt  r2, r1, -1024 ; nao salta
  26 | 00000101 00100000 10001101 | jalr r3, r0, func  ; func()
  27 | 00001001 11100000 00000101 | addi r1, r0, 79    ; r1 = 79
  28 | 00000000 00000000 01000010 | halt
  29 |                            | func:
  29 | 00001100 00100000 00000101 | addi r1, r0, 97    ; r1 = 97
  2a | 00000000 00000110 10000001 | jalr r0, r3, 0     ; retorna
```

Código de teste

```
v3.0 hex bytes plain big-endian
026005008009004204002444004 2c4004304006005004344004384004 3c4026005014305010345006385 01
e3c5dead070202c5beef0700200200000 5000045cafe070202c5002006002005004009002 30affe0058043
0200434affe00580234200258affe0058043820043caffe0058025c205208d09e0050000420c2005000681
```

Dump da memória de instruções com o programa teste

Depois de carregar o código na memória de instruções, é possível verificar se o processador está se comportando adequadamente observando a saída R1 do banco de registradores e a memória de dados.