


# ISA e Microarquitetura

Helena Nocera GRR: 20246462

## ISA

Green Card:

T <sub>r</sub>	opcode	Mnemonic		Tipo	Nome	Operação
0000		add	R	Add		$R[ra] = R[ra] + R[rb]$
0001		sub	R	Sub		$R[ra] = R[ra] - R[rb]$
0010		mul	R	Mult		$R[ra] = R[ra] * R[rb]$
0011		div	R	Divide		$R[ra] = R[ra] / R[rb]$
0100		sll	R	Shift Left Logical		$R[ra] = R[ra] \ll R[rb]$
0101		slr	R	Shift Right Logical		$R[ra] = R[ra] \gg R[rb]$
0110		and	R	And		$R[ra] = R[ra] \& R[rb]$
0111		or	R	Or		$R[ra] = R[ra]   R[rb]$
1000		xor	R	Xor		$R[ra] = R[ra] \wedge R[rb]$
1001		addi	I	Add Immediate		$R[ra] = R[ra] + Imm$
1010		lw	R	Load Word		$R[ra] = M[rb]$
1011		sw	R	Store Word		$M[rb] = R[ra]$
1100		jal	I	Jump and link		$Pc += Imm \quad R[ra] = PC += Imm$
1101		jalr	R	Jump and link Register		$Pc += R[rb] \quad R[ra] = PC += R[rb]$
1110		brzr	R	Branch on Zero Register		$if(R[ra] == R[0]) \quad Pc += R[rb]$
1111		not	R	Not		$R[ra] = !R[rb]$

Tipo R												
Bits	11	10	9	8	7	6	5	4	3	2	1	0
	Opcod e				Ra				Rb			

Tipo I												
Bits	11	10	9	8	7	6	5	4	3	2	1	0
	Opcod e				Ra				Imm			

Essa é uma ISA de 12 bits com dois modelos de instrução do tipo R em que os 4 bits mais significativos são o opcode, nos próximos 4 bits recebe o registrador R[ra] e nos últimos 4 bits recebe o registrador R[rb]. Esse formato serve para operações que operam com dois registradores, já o outro modelo de instrução I é para usar valores imediatos. Ele tem os 4 últimos bits o opcode, os 4 anteriores um registrador R[ra] e nos primeiros 4 bits um valor imediato em complemento de 2, ou seja, ele só aceita valores entre -8 e 7 e para alcançar valores maiores é necessário salvar os valores imediatos várias vezes no registrador.

Como temos 4 bits para endereçar os registradores, escolhi fazer 16 registradores sendo o r0 uma constante 0. Esta é uma ISA de 12 bits com dois modelos de instrução. O tipo R tem os 4 bits mais significativos como opcode, os 4 bits seguintes para o registrador R[ra], e os últimos 4 bits para o registrador R[rb]. Este formato é usado para operações que envolvem dois registradores. Já o outro modelo, tipo I, é para usar valores imediatos. Ele tem os 4 últimos bits para o opcode, os 4 anteriores para um registrador R[ra], e os primeiros 4 bits para um valor imediato em complemento de 2. Isso significa que ele só aceita valores entre -8 e 7; para alcançar valores maiores, é necessário salvar os valores imediatos várias vezes no registrador.

Como temos 4 bits para endereçar os registradores, optei por usar 16 registradores, sendo r0 uma constante 0.

## Registradores:

Convenção:

x0: nulo

x1 - x7: temporário

x8 - x10: salvo

x11 - x14: argumento

x15: ra

# Implementação

A implementação foi feita através do Logisim-Evolution.

O PC foi feito com um registrador e um *multiplexer* (MUX) que contém três opções de seleção: somar mais um ao valor do PC (instrução feita na maioria das vezes), somar ao valor do PC um número imediato para realizar saltos com a instrução `Jal`, e somar o valor de um registrador ao PC para realizar saltos com as instruções `jalr` e `brzr`.

As instruções estão salvas em uma memória ROM, pois, uma vez escritas, não precisam mais ser alteradas, sendo necessária apenas a leitura delas.

O banco de registradores foi implementado com 15 registradores e um valor constante de 0. Ele possui a opção de habilitar ou desabilitar a escrita e, caso haja escrita, pode ser o resultado da ULA, um valor resgatado da memória através do `Load Word`, o valor resultante de `PC += imm` ou `PC += registrador`.

A ULA realiza instruções lógicas e aritméticas com os valores recebidos. O primeiro valor será sempre `R[ra]` e o segundo varia entre `R[rb]` e `Imm`. A instrução `brzr` não está dentro da ULA; ela é feita com um comparador que retorna 1 caso o número seja igual a 0. Depois, é conectada com uma porta OU para saber se vai conectar o `PC += registrador` ao PC, pois, além de `brzr`, o `jalr` também usa esse modelo.

A memória é uma memória RAM que lê a posição de `R[rb]` e escreve o conteúdo de `R[ra]`, ou lê a posição `R[rb]` e escreve em `R[ra]`.

A unidade de controle vai decidir, em cada instrução, quais serão as entradas de cada MUX.

	pc_sel	wbreg	mux_ula	dado_breg	wr	op_ula	pcreg		
add	00	1	0	00	0	0001	0	001000000010	202
sub	00	1	0	00	0	0010	0	001000000100	204
mul	00	1	0	00	0	0011	0	001000000110	206
div	00	1	0	00	0	0100	0	001000001000	208
sll	00	1	0	00	0	0101	0	001000001010	20a
slr	00	1	0	00	0	0110	0	001000001100	20c
and	00	1	0	00	0	0111	0	001000001110	20e
or	00	1	0	00	0	1000	0	001000010000	210
xor	00	1	0	00	0	1001	0	001000010010	212
addi	00	1	1	00	0	0001	0	001100000010	302
lw	00	1	0	01	0	0000	0	001001000000	240
sw	00	0	0	00	1	0000	0	000000100000	20
jal	01	1	0	10	0	0000	0	011010000000	680
jalr	10	1	0	01	0	0000	1	101001000001	a41
brzr	10	0	0	11	0	0000	0	100011000000	8c0
not	00	1	0	00	0	1010	0	001000010100	214

A implementação da instrução LW não funcionou tão bem, pois ela necessita de duas batidas do clock para resgatar o valor correto. A forma que eu contornei isso foi quando programar escrever a mesma instrução LW duas vezes.

PC:

1 registrador

- 2 multiplexadores
- 3 somadores

Banco de registradores:

- 15 registradores
- 17 multiplexadores
- 1 demultiplexador
- 1 constante 0

ULA:

- 1 somador
- 1 subtrator
- 1 multiplicador
- 1 divisor
- 2 deslocadores
- 1 porta and
- 1 porta or
- 1 porta xor
- 1 porta not
- 1 multiplexador

## Tradução do código

Para passar o código de assembly para binário eu escrevi um código em python que recebe um documento e transforma cada linha em seu respectivo binário, para isso eu informei o opcode de cada instrução e os dois tipos de instruções.

Depois que já estava em binário eu usei o código em python usado no vídeo de explicação da unidade de controle.