Processador Uniciclo

Eduarda Saibert Professor Daniel Oliveira

Novembro de 2023

1 Circuito Geral

O trabalho a seguir diz respeito à um processador uniciclo, que respeita as seguintes instruções RISC-V (tabela 1):

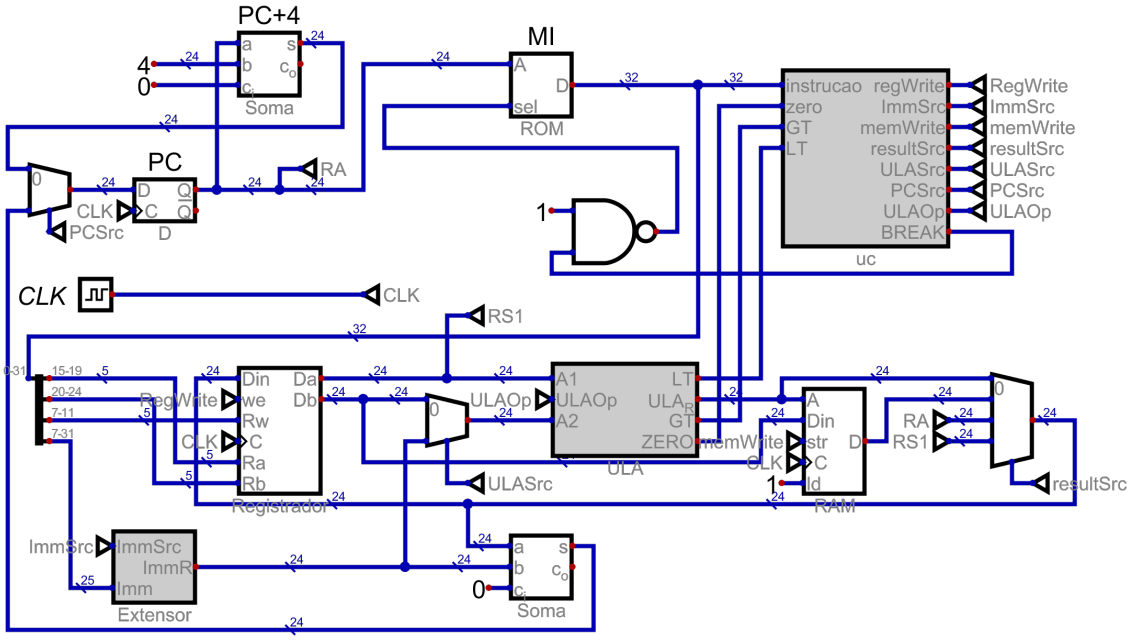
Tabela 1 – Tipos de Instrução

Função	**CodOp**	**Tipo**	**Semântica**
ADD	0110011	R	rd = rs1 + rs2
SUB	0110011	R	rd = rs1 - rs2
XOR	0110011	R	rd = rs1 ĥs2
OR	0110011	R	rd = rs1 | rs2
AND	0110011	R	rd = rs1 & rs2
SLL	0110011	R	rd = rs1 « rs2
SLT	0110011	R	rd = (rs1 < rs2) ? 1:0
ADDI	0010011	I	rd = rs1 + imm
XORI	0010011	I	rd = rs1 îmm
ORI	0010011	I	rd = rs1 | imm
ANDI	0010011	I	rd = rs1 & imm
SLLI	0010011	I	rd = rs1 « imm[0:4]
SLTI	0010011	I	rd = (rs1 < imm) ? 1:0
BEQ	1100011	B	if (rs1 == rs2) PC += imm
BNE	1100011	B	if (rs1 != rs2) PC += imm
BLT	1100011	B	if (rs1 < rs2) PC += imm
BGE	1100011	B	if (rs1 >= rs2) PC += imm
LW	0000011	I	rd = M[rs1+imm][0:31]
SW	0100011	S	M[rs1+imm][0:31] = rs2[0:31]
JAL	1101111	J	rd = PC+4 PC += imm
JALR	1100111	I	rd = PC+4 PC = rs1 + imm
EBREAK	1110011	I	PC = PC + 0

Dito isso, adaptou-se o RISC-V para operações de 24 bits, visto que o simulador de circuitos digitais *Digital* não opera ROMs ou RAMs com 32 bits.

A seguir, encontra-se o circuito geral do processador, ilustrado pela figura 1

Figura 1 – Circuito
Arquivo: pc+mi



2 Unidade de Controle

A Unidade de Controle foi construída de forma intuitiva a partir das etapas: i) decodificar o tipo de instrução; ii) gerar saídas de controle refentes à instrução; iii) gerar tipo de operação da ULA. Para simplificar o processo, foi criada a tabela 2.

Em primeiro momento, por meio do código de operação da instrução, a unidade de controle i) decodifica o tipo de instrução (figura 2).

Figura 2 – Tipo de Função
Arquivo: qualehfuncao

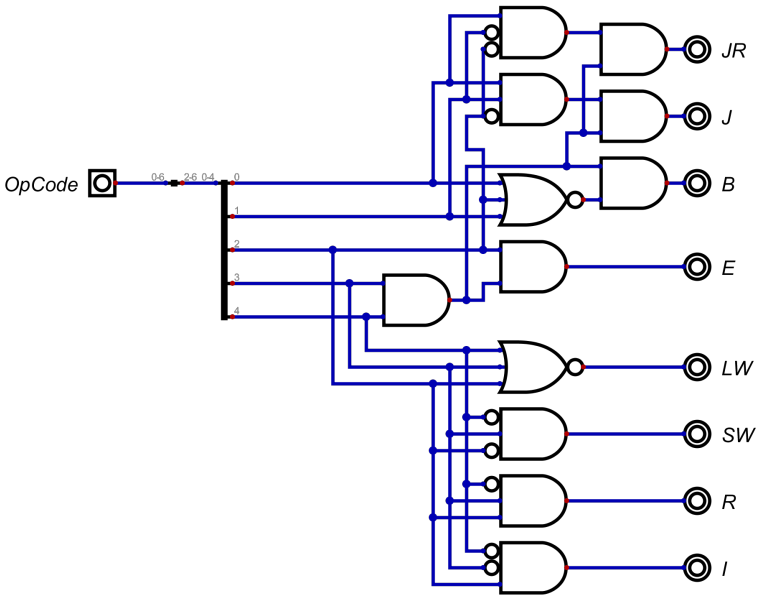
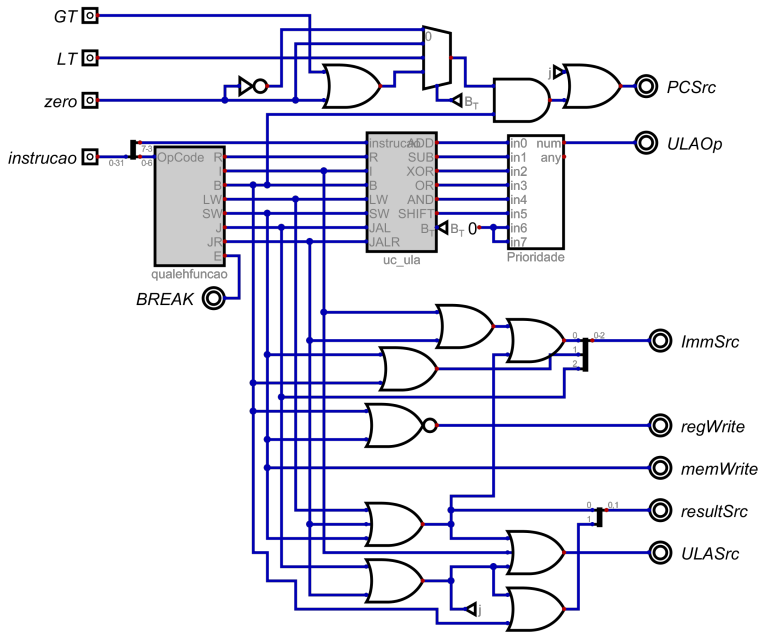


Tabela 2 – UC

Função	Func3	Func7	RegWrite	ImmSrc	ULAOp		MemWrite	ResultSrc	ULASrc	PCSrc
ADD	0x0	0x00	1	000	ADD	000	0	00	0	0
SUB	0x0	0x20	1	000	SUB	001	0	00	0	0
XOR	0x4	0x00	1	000	XOR	010	0	00	0	0
OR	0x6	0x00	1	000	OR	011	0	00	0	0
AND	0x7	0x00	1	000	AND	100	0	00	0	0
SLL	0x1	0x00	1	000	SHIFT	101	0	00	0	0
SLT	0x2	0x00	1	000	SUB	001	0	00	0	0
-	-	-	-	-	-	-	-	-	-	-
ADDI	0x0	0x00	1	001	ADD	000	0	00	1	0
XORI	0x4	0x00	1	001	XOR	010	0	00	1	0
ORI	0x6	0x00	1	001	OR	011	0	00	1	0
ANDI	0x7	0x00	1	001	AND	100	0	00	1	0
SLLI	0x1	0x00	1	001	SHIFT	101	0	00	1	0
SLTI	0x2	0x00	1	001	SUB	001	0	00	1	0
-	-	-	-	-	-	-	-	-	-	-
BEQ	0x0	0x00	0	010	SUB	001	0	00	0	1
BNE	0x1	0x00	0	010	SUB	001	0	00	0	1
BLT	0x4	0x00	0	010	SUB	001	0	00	0	1
BGE	0x5	0x00	0	010	SUB	001	0	00	0	1
-	-	-	-	-	-	-	-	-	-	-
LW	0x2	0x00	1	001	ADD	000	0	01	1	0
SW	0x2	0x00	0	011	ADD	000	1	01	1	0
-	-	-	-	-	-	-	-	10	-	-
JAL	0x0	0x00	0	100	X		0	11	1	1
JALR	0x0	0x00	1	001	ADD	000	0	01	1	1
EBREAK										

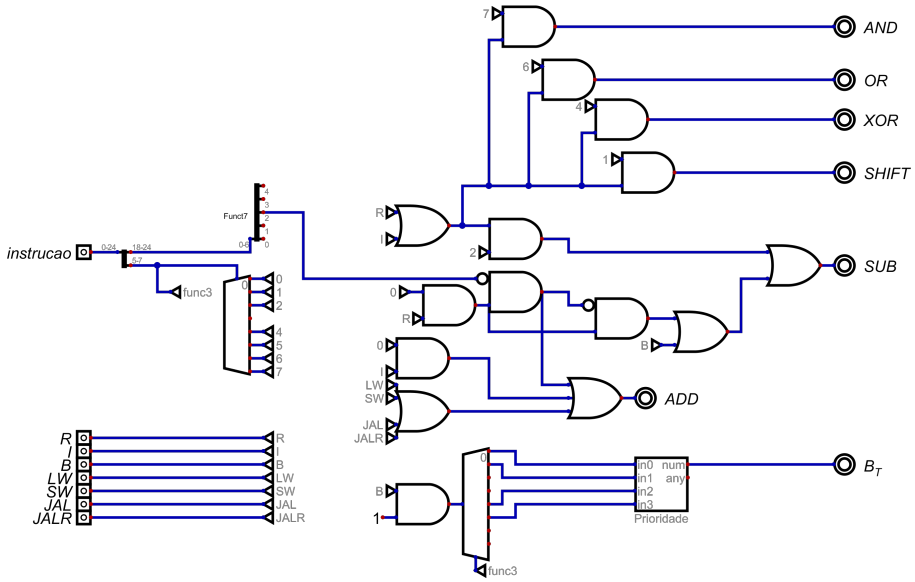
Logo após, ii) gera a maior parte das saídas de controle (RegWrite, ImmSrc, MemWrite, ResultSrc, ULASrc, PCSrc) (figura 3). Em específico, a saída PCSrc necessita da confirmação do tipo *jump* ou *callback* da ULA para efetuar as instruções de *branch*.

Figura 3 – Saídas
Arquivo: uc



Por fim, a instrução é encaminhada para a segunda parte da unidade de controle, que fornece a saída referente a ULA (ULAOp). Nesse componente, as partes *funct3* e *funct7* são utilizadas para diferenciar instruções com mesmo código de operação (figura 4). A saída *Bt* codifica o tipo de *branch*, para facilitar a verificação de mudanças no fluxo de execução.

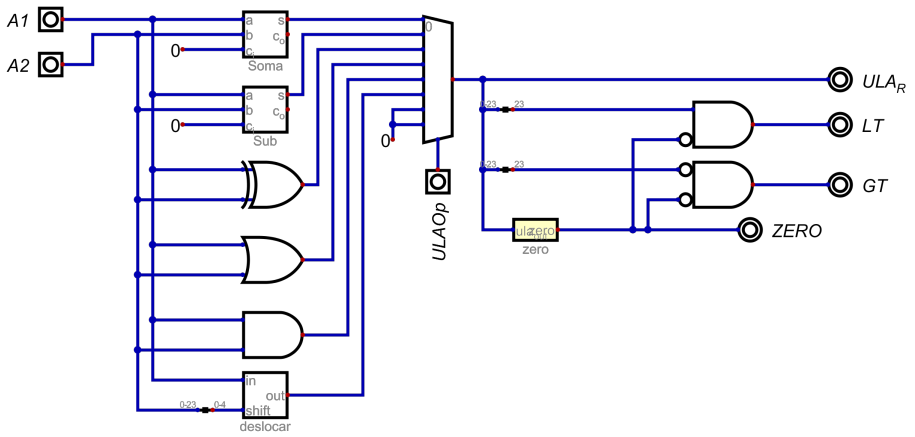
Figura 4 – Ula UC
Arquivo: uc_ula



3 ULA

As operações da ULA foram escolhidas conforme tabela 2. Além disso, a ULA é responsável pelas saídas ZERO, GT (*greater or equal than*) e LT (*less than*), úteis para a operação de subtração no caso de *branches*. O circuito pode ser visualizado na figura 5.

Figura 5 – ULA
Arquivo: ula



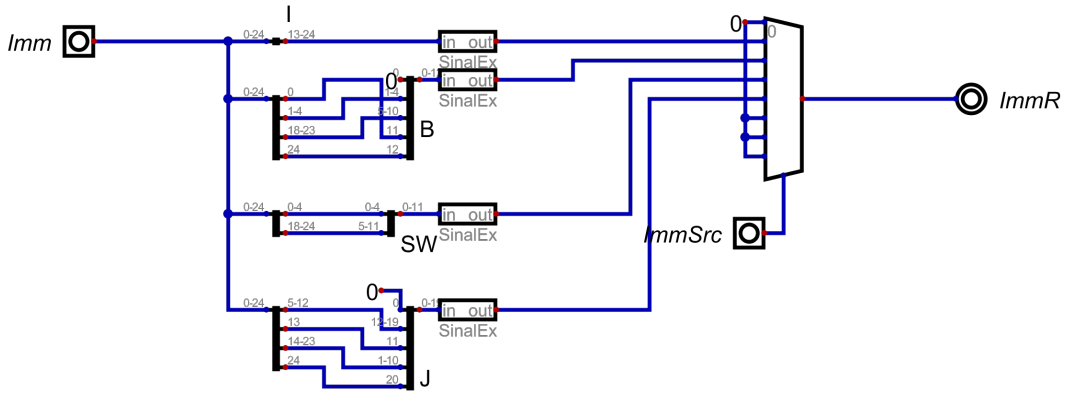
4 Imediatos

Os imediatos em RISC-V seguem a implementação disponível na tabela 3. Dito isso, no que se refere ao trabalho, o componente de extensão para cada tipo de imediato (definidos na tabela 2) está presente na figura 6

Tabela 3 – Imediatos RISC-V

31-25	24-20	19-15	14-12	11-7	6-0	Tipo
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[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Figura 6 – Imediatos Implementação
Arquivo: IMMSRC



5 Assembly

O processador deve operar o seguinte código fibonacci, transcrito de C para Assembly:

```
    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];
    }
}

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

Visto isso, o código final (introduzido à ROM do processador):

```
addi s1, zero, 20   # s1 tem o valor de tam
addi a0, zero, 0    # a0 tem o endereco do vetor (0)
addi a1, s1, 0   # a1 tem o valor de tam
addi a2, zero, 1      # a2 tem o primeiro elem da sequencia
addi a3, zero, 1   # a3 tem o segundo elem da sequencia
jal fib       # pula para a funcao
ebreak       # encerra o programa

fib:
sw a2, 0(a0)   # guarda 1 no primeiro espaco do vetor
sw a3, 4(a0)   # guarda 1 no segundo espaco do vetor
addi t1, zero, 2   # int i = 2

for:
bge t1, a1, fora_for  # sai do for se i >= tam
slli t2, t1, 2   # t2 = i*4
add t2, t2, a0

lw t3, -4(t2)   # carrega o valor de v[i-1]
lw t4, -8(t2)   # carrega o valor de v[i-2]

add t5, t3, t4   # realiza a soma de v[i-1] e v[i-2]
sw t5, 0(t2)   # guarda o valor de v[i]
addi t1, t1, 1   # incrementa 1 em i
jal x0, for
fora_for:

jalr ra, 4       # volta para main
```