Rafael Ribeiro Kluge

GRR20244439

Projetos Digitais e Microprocessadores

Trabalho Final

2024/2

ISA do RISC-V

Tipo/Bits	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R						RS2				RS1				RD				FUNC3			OPCODE			
I	IMM									RS1				RD				FUNC3			OPCODE			
S & B	IMM									RS1				RS2				FUNC3			OPCODE			
J	IMM													RD				IMM			OPCODE			

Tipo	Instrução	Nome	FUNC3	OPCODE	Descrição
R	ADD	Addition	000	0000	add rd, rs1, rs2
	SUB	Subtraction	001		sub rd, rs1, rs2
	MUL	Multiplication	010		mul rd, rs1, rs2
	DIV	Division	011		div rd, rs1, rs2
	SLL	Shift Left Logical	100		sll rd, rs1, rs2
	SRL	Shift Right Logical	101		slt rd, rs1, rs2
I	ADDI	ADD Immediate	000	0001	addi rd, rs1, imm
	SLLI	Shift Left Logical Imm	100		slli rd, rs1, imm
	SRLI	Shift Right Logical Imm	101		slti rd, rs1, imm
	LW	Load Word	000	0011	slti rd, rs1, imm
	JALR	Jump And Link Reg	000	0110	jalr rd, rs1, imm
B	BEQ	Branch Equal	000	0010	beq rs1, rs2, imm
	BNE	Branch Not Equal	001		bne rs1, rs2, imm
	BLT	Branch Less Than	010		blt rs1, rs2, imm
	BGE	Branch Greater or Equal	011		bge rs1, rs2, imm
S	SW	Store Word	000	0100	sw rs1, rs2, imm
J	JAL	Jump And Link		0101	jal ra, imm

1. Introdução

Esse trabalho consiste na implementação da arquitetura de um conjunto de instruções de 24 bits (ISA) do RISC-V. Essa implementação foi baseada nas instruções do projeto, que permite escolher as instruções de implementação bem como as funções que são testadas.

2. Instruções

Foram implementadas 17 instruções nesse trabalho (entrada e saída de 24 bits), armazenadas na memória de instruções, que suportam as estruturas básicas de programação, como condições e desvios. São elas:

- Instruções Aritméticas sem imediato

  ADD: Adição
  SUB: Subtração
  MUL: Multiplicação
  DIV: Divisão
  SLL: Desvio à esquerda lógico
  SLT: Desvio à direita lógico

- Instruções Aritméticas com imediato

  ADDI: Adição com imediato
  SLLI: Desvio à esquerda lógico com imediato
  SLTI: Desvio à direita lógico com imediato

- Instruções de Acesso à Memória

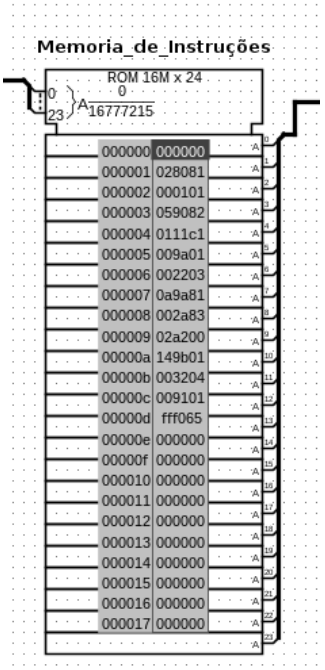
  LW: Carrega palavra
  SW: Salva palavra

- Instruções de comparação

  BEQ: Se há igualdade
  BNE: Se há desigualdade
  BLT: Menor que
  BGE: Maior ou igual que

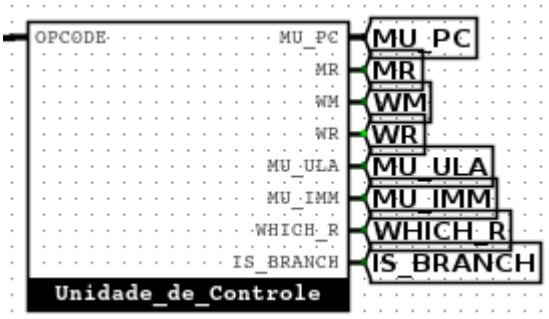
- Instruções de salto

  JAL: Desvio direto
  JALR: Desvio para endereço



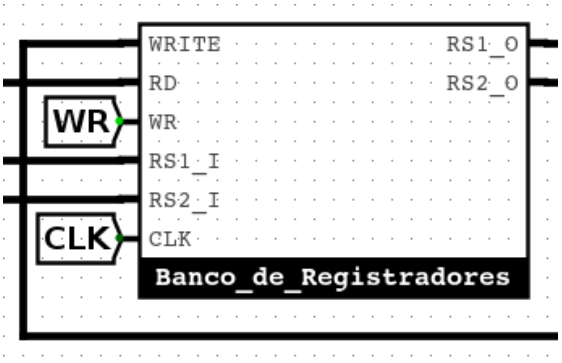
3. Unidade de Controle

A unidade de controle é uma parte fundamental do processador. É uma memória ROM e nela, nós decidimos a partir de determinada instrução, quais ações serão feitas. Então, a partir do *opcode* recebido, e com essa unidade codificada, é feita a manipulação sobre quais partes do circuito vão executar. Suas funções, de maneira especifica, serão explicitadas ao longo da descrição.



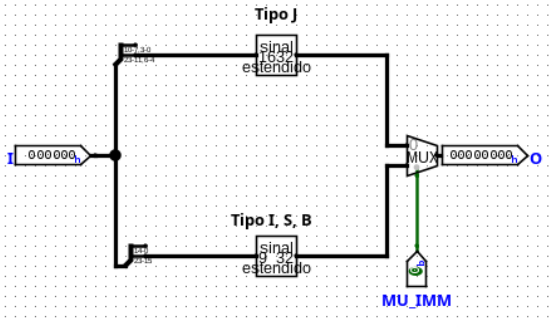
4. Banco de Registradores

O banco de registradores é uma unidade com 6 entradas e 2 saídas. Ele escolhe quais bits da memória de instruções vão para as entradas *RS1_I*, *RS2_I* e *RD*. Dependendo da instrução, não há o *RD* ou *RS2*, mas isso é manipulado pela unidade de controle (*WHICH_R*), assim como se é preciso escrever em um registrador ou não (*WR*). Também, há uma entrada *WRITE*, que é o valor definitivo a ser escrito e o *CLOCK*. Sua saída se consiste no conteúdo dos registradores que foram selecionados (*RS1_O, RS2_O*). O presente banco possui 16 registradores, porém os programas que serão testados foram divididos, então usaremos apenas parte deles.



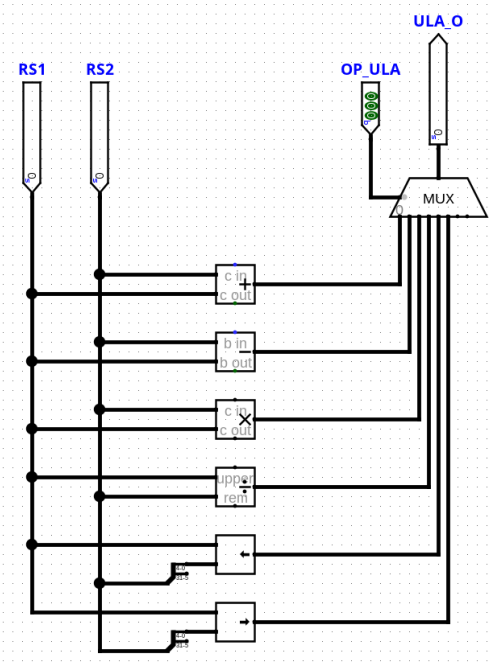
5. Processador de Imediato

Em determinadas instruções, não temos a necessidade de utilizar *RS2* mas sim um valor *IMM,* como exemplo instruções do tipo I. Juntamente com isso, precisamos iterar o *PC*. Como a nossa *ULA* (Unidade Lógica Aritmética) recebe instruções de 32 bits, e os valores imediatos sempre terão apenas parte dos bits da instrução, precisamos estendê-los para chegar aos bits necessários. Para isso, utilizamos um processador de imediato, que é controlado por um *MU_IMM* (na unidade de controle), diferenciando as instruções e extensões dos imediatos



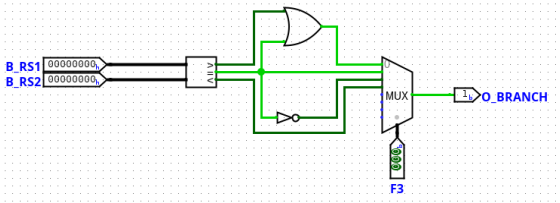
5. Unidade Lógica Aritmética (ULA)

A ULA, como diz o próprio nome, fará operações aritméticas como: ADD, SUB, MUL, DIV, SLL, SLR, ADDI, SLLI e SLRI, a qual tem duas entradas de 32 bits (*RS1* e *RS2*), uma entrada *OP_ULA* que será o valor de *FUNC3*, que diferencia as funções, e uma saída de 32 bits com o valor calculado. Ainda, há um *MUX* com um seletor *MU_ULA*, controlado pela unidade de controle e servindo para diferenciar se a função necessita de um *RS2* ou um *IMM*.



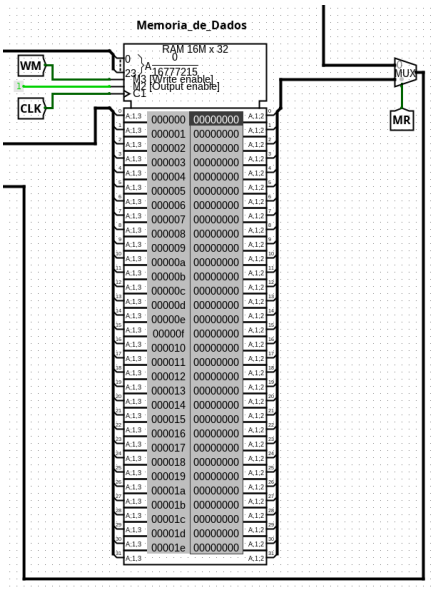
6. Unidade Branch

A unidade branch é onde serão feitas as operações comparativas como: BEQ, BNE, BLT e BGE, sendo controladas pelo *FUNC3*, que determinada qual operação será realizada. Além da entrada de *FUNC3*, há duas entradas de 32 bits (*B_RS1* e *B_RS2*) e uma saída de apenas 1 bit (*O_BRANCH*), que dirá se a operação escolhida é verdadeira ou falsa.



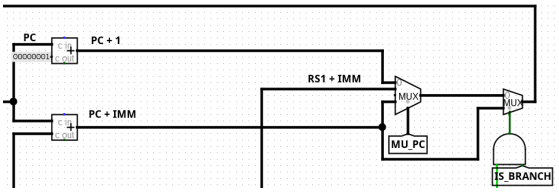
7. Memória de Dados

A memória de dados é uma RAM que tratará de operações de acesso à memória como: LW e SW. Alem de possuir uma entrada de 24 bits, que será o resultado da ULA, possui 3 entradas de um bit, sendo *WM* para definir se haverá escrita na memória, *Output Enable* para deixar habilitado a saída e também o *CLOCK*. Sua saída possui 32 bits e é ligada à um *MUX* que decide se o resultado será escrito no registrador (*MR*) ou não. As operações *WM* e *MR* são controladas pela unidade de controle.



8. Controle do PC

O PC (*program counter*) tem seu valor manipulado dado por dois somadores e dois *MUX*. No primeiro *MUX*, chegam valores como: *PC+1* (como instrução padrão), *PC+IMM* (caso há saltos de imediato) e *RS1+IMM* (caso haja a instrução específica *JALR*). As anteriores são controladas por *MU_PC* pela unidade de controle. O segundo *MUX* é controlado pelo resultado de uma porta *AND* entre o resultado de *O_BRACH* e *IS_BRANCH*, que funciona como uma segunda etapa, em virtude de precisar desviar o fluxo caso a comparação seja verdadeira ou não caso contrário.



Códigos em assembly

Fibonacci:

```
        addi t0, zero, 1
        addi t1. zero, 1
        addi t2, zero, 3
loop:
        blt a0, t2, fim_loop
        add t3, t0, t1
        addi t0, t1, 0
        addi t1, t3, 0
        addi a0, a0, -1
        jal zero, loop
fim_loop:
        addi a0, t1, 0
```

Soma de vetores:

```
        addi t0, zero, 5
        addi t1, zero, 0
loop:
        beq t1, t0, fim_loop
        slli t2, t1, 2
        addi t3, t2, A
        lw t3, t3, 0
        addi t4, t2, B
        lw t4, t4, 0
        add t3, t3, t4
        addi t5, t2, C
        sw t5, t3, 0
        addi t1, t1, 1
        jal zero, loop
fim_loop:
```

A = {1, 2, 3, 4, 5}
B = {10, 20, 30, 40, 50}

Testa o restante:

```
        addi t0, zero, 3
        addi t1, zero, 16
        addi t2, zero, 15
        addi t3, zero, 2
        sub t4, t1, t0
        mul t5, t0, t2
        div t6, t2, t0
        sll t7, t1, t3
        srl t8, t1, t3
        slri t9, t1, 3
        bne t9, t3, fora
        beq t8, t6, fora
        addi t8, t8, 1
        jalr zero, t9, 0
fora:
```

Passo a Passo para transformação do assembly em código de máquina

A partir do *greencard*, que contém a divisória dos bits dentro de um espaço de 24 bits, assim como o *opcode* das funções e *func3* que foram as primeiras definições e o primeiro passo desse projeto, basta encaixar os bits certos nos dois últimos. Para registradores, basta verificar a ordem em que estão conectados no banco de registradores, que será a ordem usual crescente, atentando-se que essa ordenação é na base binária. Para imediatos, como exemplo, se quero somar o valor 3, basta convertê-lo para binário (11) e logo após encaixar nos bits de imediato (que podem ser 9 ou 16), completando com zeros à esquerda. Por fim, para colocar no circuito, tanto na memória de instruções como memória de dados, basta converter os dados de binário para hexadecimal, que é a base que o *Logisim Evolution* aceita.

Códigos em binário

Fibonacci

```
        00000000100000010000001
        00000000100000100000001
        00000001100000110000001
        00000010100000100000001
loop:
        00000011001000011010010
        00000001000010101000000
```

000000000001000010000001
000000000010100100000001
111111111010001000000001
111111111111100000110101

fim_loop:
000000000001001000000001

Soma de vetores:

000000101000000010000001
000000000000000100000001

loop:
000001011001000010000010
000000010001000111000001
000000001001101000000001
000000000010001000000011
000010101001101010000001
000000000010101010000011
000000101010001000000000
000101001001101100000001
000000000011001000000100
000000001001000100000001
111111111111000001100101

fim_loop:

Testa o restante:

000000011000000010000001
000010000000000100000001
000001111000000110000001
000000010000001000000001
000000001001001010010000
000000011000101100100000
000000001001101110110000
000000100001010001000000
000000100001010011010000
000000011001010101010001
000000100101001000010010
000000100100101110110010
000000001100110010000001

fora:

Listagem de componentes utilizados

- PC: 1 registrador de 32 bits
- Memória de Instruções: 1 memória ROM de entrada 24 bits e saída 24 bits.
- Banco de Registradores: 16 registradores de 32 bits.
- Processador de IMM: 2 extensor de bits, um de entrada 16 bits e outro de entrada 9 bits. Ambos com saída de 32 bits.
- Unidade de Controle: 1 memória ROM de entrada 4 bits e saída 16 bits.
- Branch: 1 comparador de 32 bits; 1 mux com 3 bits para seleção.
- ULA: 1 somador; 1 subtrator, 1 multiplicador; 1 divisor; 2 deslocadores (esquerda e direita). Todos de 32 bits.
- Memória de Dados: 1 memória RAM de entrada 24 bits e saída 32 bits.
- Complementares: 10 multiplexadores; 1 demultiplexador; 2 somadores; 1 porta and.

Convenção

A convenção depende muito da função que é testada. Para a função fibonacci, os registradores x0 à x2 equivalem a t0 à t2, e x3 equivale a a0. Para soma de vetores, x0 à x5 equivale a t0 à t5 e para o último, x0 à x9 é t0 à t9. Não foram necessários registradores salvos.