Trabalho 2 - Arquitetura Redux-V

EDUARDO KALUF GRR20241770

May 3, 2025

1 Apresentação

O objetivo deste trabalho é a implementação da **ISA** *Redux-V*, utilizando as ferramentas **EGG** e **Logisim**, visando executar o programa Assembly desenvolvido no "Trabalho 1", além da criação de três instruções adicionais próprias, visando otimizar o problema apresentado. Esta **ISA** é baseada na arquitetura *Risc-V*, porém refatorada a fim de ser mais compacta e amigável para quem está iniciando no mundo da arquitetura de computadores.

Dessa forma, o tamanho da palavra é de oito bits, com endereçamento byte a byte, arquitetura **Load-Store** e quatro registradores disponíveis. Dos oito bits, quatro são utilizados para o **OPCODE**, permitindo um total de dezesseis instruções para serem exploradas, as quais serão discutidas mais adiante. Além disso, por padrão, existem dois formatos de instrução:

Formato Padrão 1: Tipo I

Tipo I								
Bits	7	6	5	4	3	2	1	0
	Opcode				**Imm.**			

Formato Padrão 2: Tipo R

Tipo R								
Bits	7	6	5	4	3	2	1	0
	Opcode				**Ra**		**Rb**	

2 Motivação e Arquitetura

A proposta do "Trabalho 1" consistia em criar um programa em *Redux-V* que preenchesse um vetor com números pares (0 a 18) e outro com números ímpares (1 a 19), para posteriormente somar ambos e armazenar os resultados em um terceiro vetor. Durante o desenvolvimento do código Assembly no primeiro trabalho, as limitações da arquitetura trouxeram diversos desafios, especialmente devido ao número reduzido de registradores e ao tamanho limitado dos imediatos.

Dessa forma, ao implementar o "Trabalho 2", tive como objetivo principal facilitar ao máximo a tarefa do programador, evitando a necessidade de programar loops. Além disso, busquei dar caminhos para que o programa pudesse ser mais eficiente, sucinto e organizado.

Com base nesses objetivos, elaborei as seguintes instruções.

2.1 Increment

A primeira instrução que implementei foi a instrução "Increment", visando um propósito amplo e genérico. No entanto, busquei torná-la o mais útil possível para o usuário, não limitando sua função apenas ao incremento de 1 em algum registrador específico, por exemplo.

Para isso, desenvolvi um novo formato de instrução, estabelecendo também um comportamento especial ao utilizar o registrador **R0**. Essa decisão foi tomada porque incrementar diretamente o **R0** não teria muito sentido, visto que a instrução addi já consegue cumprir essa tarefa de maneira até mais eficiente.

Formato Novo 1: Formato para instrução Increment

Tipo INC								
Bits	7	6	5	4	3	2	1	0
	Opcode				**Ra**		**Imm Unsigned**	

Esta instrução recebe um registrador e um imediato sem sinal, incrementando o valor contido nesse registrador. No entanto, caso o registrador **R0** seja informado, o incremento ocorrerá em todos os registradores, incluindo o próprio **R0**. Dessa forma, aproveita-se o que seria um comando duplicado para aprimorar ainda mais a arquitetura.

2.2 Load Vector

A instrução "Load Vector" vem para resolver o processo de preenchimento dos vetores e, assim como a "Increment", possui algumas características especiais. Para que possa ser utilizada corretamente, é necessário que alguns registradores sejam configurados de antemão com valores específicos, seguindo a lógica abaixo:

- R0 → Endereço de memória do início do vetor;

- R1 → Incremento de memória que o vetor receberá a cada execução;
- R2 → Primeiro valor a ser inserido no vetor;
- R3 → Incremento que o valor de R2 receberá a cada inserção.

Ademais, ela recebe o tamanho do vetor como parâmetro e utiliza um novo formato de instrução:

Formato Novo 2: Formato para instruções Vetoriais

Tipo V								
Bits	7	6	5	4	3	2	1	0
	Opcode				**V-SIZE (Unsigned)**			

Vale ressaltar que o tamanho final do vetor será $V\_SIZE \times R1$, e ao fim da execução, $R0$ estará posicionado em $R0 + (R1 \times V\_SIZE)$.

Exemplo de funcionamento:

- R0 = 27
- R1 = 1
- R2 = 5
- R3 = 10

loadv 4

0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
···	5	15	25	35	···

2.3 Add Vector

Juntamente com a instrução vetorial "Load Vector", criei também a instrução "Add Vector", responsável por somar dois vetores e armazenar o resultado dessa soma em um terceiro. Essa instrução utiliza o mesmo formato da "Load Vector", porém os registradores têm funções diferentes, conforme descrito a seguir:

- R0 → Endereço de memória do início do primeiro vetor a ser somado;
- R1 → Endereço de memória do início do segundo vetor a ser somado;
- R2 → Endereço de memória do início do vetor que armazenará o resultado da soma;
- R3 → Não possui função nesta operação. Porém, a instrução não garante que o registrador mantenha seu estado inicial;

Além disso, é necessário que os vetores possuam o mesmo tamanho, pois a soma será feita posição por posição.

Exemplo de funcionamento:

- R0 = 2
- R1 = 5
- R2 = 8
- R3 = 3

addv 3

0x02	0x03	0x04
3	1	2

0x05	0x06	0x07
2	4	6

0x08	0x09	0x0A
5	5	8

2.4 Resultado

Desta forma, após a implementação dessas três novas instruções ficamos com a seguinte ISA:

Opcode	Tipo	Mnemonic	Nome	Operação
0000	R	brzr	Branch On Zero Register	if (R[ra] == 0) PC = R[rb]
0001	I	ji	Jump Immediate	PC = PC + Imm.
0010	R	ld	Load	R[ra] = M[R[rb]]
0011	R	st	Store	M[R[rb]] = R[ra]
0100	I	addi	Add Immediate	R[0] = R[0] + Imm.
0101	V	loadv	Load Vector	Micro-Code-1
0110	V	addv	Add Vector	Micro-Code-2
0111	INC	inc	Increment	Ra = Ra + Imm Unsigned
1000	R	not	Not	R[ra] = not R[rb]
1001	R	and	And	R[ra] = R[ra] and R[rb]
1010	R	or	Or	R[ra] = R[ra] or R[rb]
1011	R	xor	Xor	R[ra] = R[ra] xor R[rb]
1100	R	add	Add	R[ra] = R[ra] + R[rb]
1101	R	sub	Sub	R[ra] = R[ra] - R[rb]
1110	R	slr	Shift Left Register	R[ra] = R[ra] ¡¡ R[rb]
1111	R	srr	Shift Right Register	R[ra] = R[ra] ¿¿ R[rb]

Os microcódigos executados pelas funções são

Microcódigo 1: Load Vector

```
\begin{lstlisting}
  _loop:
    st r2, r0
    add r0, r1
    add r2, r3
    ji _loop
```

Microcódigo 2: Add Vector

```
\begin{lstlisting}
  _program2:
    ld r0, r0 ;; (r0, rx)
    ld r1, r1 ;; (r0, ry)
    add r3, r0 ;; (rx + ry -> r3)
    st r3, r2
    inc 1, r0 ;; (r0, r1, r2, r3 += 1)
    ji _program2
```

3 Assembly

Após as modificações feitas, a solução para o problema dos vetores se tornou muito mais sucinta, fácil de entender e otimizada.

Código do "Trabalho 1"

```
_setupMemoria:
  ;; Mem[-1] = (COMECO DO VETOR A) (0x70) (112)
  addi 0
  addi 7
  add r2, r0
  addi -3
  slr r2, r0
  addi -5
  st r2, r0
  ;;

  ;; Mem[-3] = COUNTER (0x0a)
  addi 6
  addi 5
  add r1, r0
  sub r0, r0
  addi -3
  st r1, r0
  ;;

  ;; Mem[-4] = 10 FIXO (0x0a)
  addi -1
  st r1, r0
  ;;

  ;; Mem[-5] = 20 FIXO (0x14)
  addi -1
  add r1, r1
  st r1, r0
  ;;

  ;; Mem[-2] = ENTRADA DO "loopSoma"
  ;; (INICIO DO VETOR (r2) N DE INSTRUCOES DO "loopSoma") (0x52)
  sub r0, r0
  addi -2
  add r3, r0
  sub r2, r1
  addi 7
  addi 5
  sub r2, r0
  st r2, r3
  ;;
```

```
_setupLoopEven:

  ;; r1 = 1 (0x01)
  sub r0, r0
  sub r1, r1
  addi 1
  add r1, r0
  ;;

  ;; r3 = (FINAL DO VETOR A) (0x79) (121)
  addi -2
  ld r0, r0
  addi 7
  addi 2
  sub r3, r3
  add r3, r0
  ;;

  ;; r2 = SAIDA DO LOOP EVEN (0x34)
  sub r0, r0
  sub r2, r2
  addi 6
  add r2, r0
  addi -3
  slr r2, r0
  addi 1
  add r2, r0
  sub r0, r0
  ;;

  ;; r0 = 20 (0x14)
  addi -5
  ld r0, r0
  ;;

_loopEven:
  addi -2 ;; r0 -= 2
  st r0, r3 ;; A[r3] = r0
  sub r3, r1 ;; r3 -= 1
  brzr r0, r2 ;; if (r0 == 0) break
  ji _loopEven

_setupLoopOdd:

  ;; r2 = SAIDA DO LOOP ODD (0x43)
  sub r2, r2
  addi 4
  slr r0, r0
  addi 3
  add r2, r0
```

```
  ;;

  ;; r3 = (FINAL DO VETOR B) (0x83) (131)
  sub r0, r0
  addi -5
  ld r0, r0
  add r3, r0
  ;;

  ;; r1 JA E IGUAL A 1 (0x01)

  ;; r0 JA E IGUAL A 20 (0x14)

_loopOdd:
  addi -1 ;; r0 -= 1
  st r0, r3 ;; B[r3] = r0
  sub r3, r1 ;; r3 -= 1
  addi -1 ;; r0 -= 1
  brzr r0, r2 ;; if (r0 == 0) break
  ji _loopOdd

_setupLoopZero:

  ;; r3 = (FINAL DO VETOR R) (0x8D) (141)
  ;; r1 = COUNTER (0x0A)
  addi -3
  ld r1, r0
  addi -2
  ld r0, r0
  add r3, r0
  ;;

  ;; r2 = SAIDA DO LOOP ZERO (POSICAO -2 da memoria) (0x52)
  sub r0, r0
  addi -2
  ld r2, r0
  ;;

_loopZero:
  sub r0, r0 ;; r0 = 0
  st r0, r3 ;; R[r3] = r0
  addi 1 ;; r0 = 1
  sub r3, r0 ;; r3 -= 1
  sub r1, r0 ;; r1 -= 1
  brzr r1, r2 ;; if (r1 == 0) break
  ji _loopZero

_loopSum:
  sub r0, r0 ;; r0 = 0
  addi -1 ;; r0 = -1
```

```
    ld r3, r0 ;; r3 = Mem[-1] (COMECO DO VETOR A)
    addi 2 ;; r0 = 1
    add r3, r0 ;; r3 += 1
    addi -2 ;; r0 = -1
    st r3, r0 ;; Mem[-1] = (POSICAO ATUAL DO VETOR A + 1)
    addi 2 ;; r0 = 1
    sub r3, r0 ;; r3 -= 1 (POSICAO ATUAL DO VETOR A)
    addi -5 ;; r0 = -4
    ld r0, r0 ;; r0 = Mem[-4] (0x0a)
    ld r1, r3 ;; r1 = A[r3]
    add r3, r0 ;; r3 += 10
    ld r2, r3 ;; r2 = B[r3]
    add r1, r2 ;; r1 = A[r3] + B[r3]
    add r3, r0 ;; r3 += 10
    st r1, r3 ;; R[r3] = A[r3] + B[r3]
    sub r0, r0 ;; r0 = 0
    addi -3 ;; r0 = -3
    ld r1, r0 ;; r1 = Mem[-3] (COUNTER)
    addi 4 ;; r0 = 1
    sub r1, r0 ;; r1 -= 1
    addi -4 ;; r0 = -3
    st r1, r0 ;; Mem[-3] = COUNTER - 1
    not r1, r1 ;; r1 = !r1 (zero a nao ser na interacao de saida)
    addi 1 ;; r0 = -2
    ld r2, r0 ;; r2 = Mem[-2] ENTRADA DO "loopSoma"
    brzr r1, r2 ;; if (r1 == 0) continue

  _loopInfinite:
    addi 0
    ji _loopInfinite
```

Código do "Trabalho 2", Otimizado com as novas instruções

```
inc r1, 1
inc r3, 2
addi 4
slr r0, r0
loadv 10
sub r2, r2
inc r2, 1
loadv 10
sub r2, r2
add r2, r0
addi -7
addi -3
sub r1, r1
add r1, r0
addi -7
addi -3
addv 10
_loopInfinite:
    addi 0
    ji _loopInfinite
```

4 Implementação e Organização

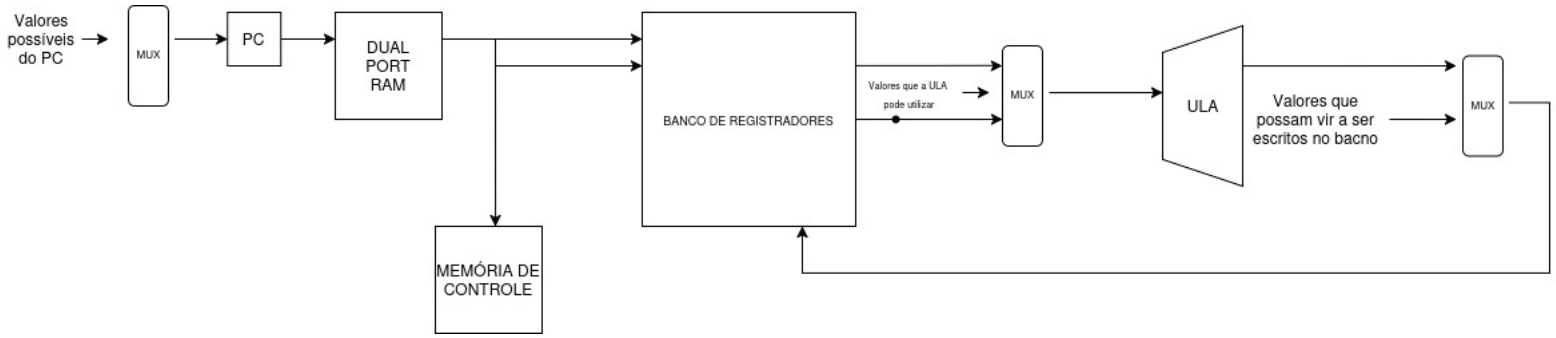
Essa seção será dividida entre a implementação padrão do *Redux-V*, a implementação da **ULA** e a implementação das três funções adicionais. Ao final, serão apresentadas a tabela de controle e informações adicionais.

4.1 Monociclo

O monociclo é simples e compacto, sendo implementado com o uso de uma **RAM** de duas portas e uma memória de controle, composta pelos seguintes módulos:

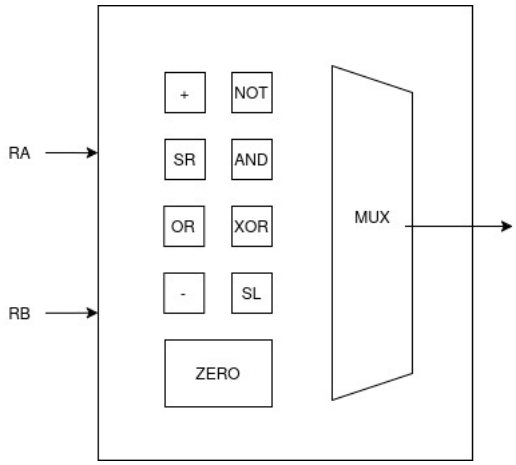
- Program Counter
- Dual-Port RAM
- Memória de Controle
- Banco de Registradores
- ULA

A seguir, apresenta-se o datapath desta arquitetura em um diagrama de blocos simplificado.



A **ULA** foi projetada para conter oito operações, além de uma saída **ZERO**, que detecta se o **RA** é nulo, utilizada em casos de branching

A seguir, apresenta-se o projeto da ULA em um diagrama de blocos simplificado.



Os códigos de controle da ULA:

Operação	OP_ULA
Not	**000**
And	**001**
Or	**010**
Xor	**011**
Soma	**100**
Subtração	**101**
Shift left lógico	**110**
Shift right lógico	**111**

4.2 Extensão

Para implementar as três instruções adicionais, além dos novos sinais de controle introduzidos no monociclo tradicional, uma nova área foi adicionada para atender às necessidades específicas dessas instruções. Essa nova área contém:

- Um novo Program Counter
- Uma nova memória de instruções
- Dois contadores

4.2.1 Increment

A instrução "Increment" opera de forma linear para os registradores **R1** a **R3**. Ela recebe um valor imediato fornecido pelo campo que normalmente indicaria o endereço do registrador **RB**, realiza a soma desse imediato com o valor do registrador **RA** na **ULA**, e então armazena o resultado de volta no próprio registrador **RA**.

Caso o registrador passado seja **R0**, o sinal de controle "INC_ALL" é ativado, habilitando a escrita em todos os registradores. Dessa forma, somadores internos ao próprio banco de registradores realizam o incremento diretamente em cada um deles.

4.2.2 Load Vector

A instrução "Load Vector" ativa o sinal de controle "LOAD_V", que altera o fluxo de operação do processador. Com isso, as instruções passam a ser buscadas a partir da memória secundária. Além disso, um contador é inicializado com o tamanho do vetor que será carregado.

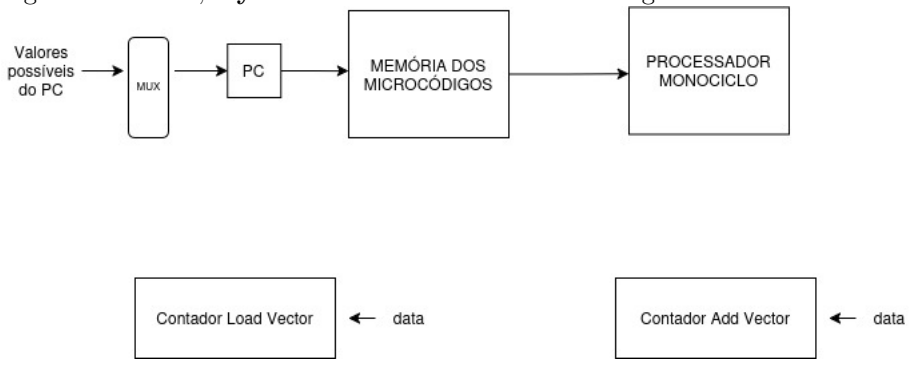
A partir disso, um Microcódigo que preenche o vetor começa a operar. Esse Microcódigo consiste em um loop simples, que finaliza com a execução de uma instrução do tipo **JI**. A cada iteração, com a execução do **JI**, o contador é reduzido em uma unidade. Quando o contador chega a zero, o processador retorna ao fluxo normal de execução.

4.2.3 Add Vector

De forma similar, a instrução "Add Vector" também modifica o fluxo do processador, ativando um contador e executando um outro Microcódigo. Contudo, há diferenças na forma como as instruções "Load" e "Add" operam dentro desse Microcódigo.

No caso da instrução "Load", o registrador **RB** passa a ser interpretado de outra forma: o **R0** passa a ser **Rx** e o **R1** passa a ser **Ry**, um par adicional de registradores utilizados exclusivamente durante este Microcódigo.

Já a instrução "Add", a operação ocorre sempre somando os valores dos registradores **Rx**, **Ry** e armazenando o resultado no registrador **R3**.



4.3 Controle

Finalmente então, temos os controles tabelados:

Bits de Controle

Mnemonic	OPCODE	ADDV	INC	LOADV	LOAD	BRZR	JI	ADDI	W_MEM	IMM_ULA	OP_ULA	W_RG	HEX_VALUE
brzr	0000	0	0	0	0	1	0	0	0	0	000	0	0x100
ji	0001	0	0	0	0	0	1	0	0	1	100	0	0x098
ld	0010	0	0	0	1	0	0	0	0	0	000	1	0x201
st	0011	0	0	0	0	0	0	0	1	0	000	0	0x020
addi	0100	0	0	0	0	0	0	1	0	1	100	1	0x059
loadv	0101	0	0	1	0	0	0	0	0	0	000	0	0x400
addv	0110	1	0	0	0	0	0	0	0	0	000	0	0x1000
inc	0111	0	1	0	0	0	0	0	0	0	100	1	0x809
not	1000	0	0	0	0	0	0	0	0	0	000	1	0x001
and	1001	0	0	0	0	0	0	0	0	0	001	1	0x003
or	1010	0	0	0	0	0	0	0	0	0	010	1	0x005
xor	1011	0	0	0	0	0	0	0	0	0	011	1	0x007
add	1100	0	0	0	0	0	0	0	0	0	100	1	0x009
sub	1101	0	0	0	0	0	0	0	0	0	101	1	0x00B
slr	1110	0	0	0	0	0	0	0	0	0	110	1	0x00D
srr	1111	0	0	0	0	0	0	0	0	0	111	1	0x00F

5 EGG e Redux-K

Juntamente com a implementação das instruções no **Logisim**, o emulador **EGG** também foi modificado para interpretá-las corretamente. Dessa forma, dou vida à arquitetura *Redux-K* (*Redux-V* versão Kaluf), que agora pode ser utilizada como qualquer outra arquitetura disponível no emulador.

REDUX-K

6 Conclusão

Com o processador finalizado, foi possível perceber novas formas de se trabalhar com o hardware, as limitações de uma **ISA** reduzida e ao mesmo tempo a ampla gama de possibilidades de melhorias e inovações que ela oferece.

Trabalhar na implementação do *ReduxV* e enfrentar o desafio da soma de vetores proporcionou diversas experiências técnicas, que me levaram por caminhos diferentes dos que estou acostumado a lidar. Isso ampliou minha visão e sensibilidade em relação ao funcionamento dos processadores e do hardware.

Foi um processo interessante e muito proveitoso.