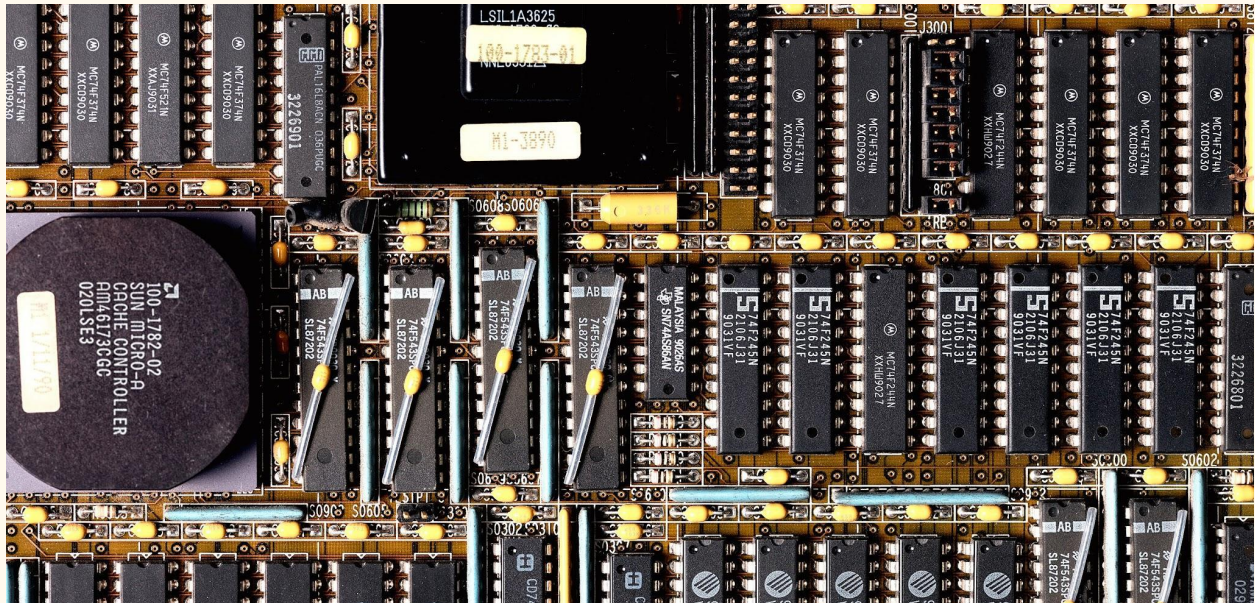


Arquitetura UFC2X - otimizações e programas

Lucas Braide Schramm de Moura - 537009

Marcelo Meireles Marques Filho - 536927



INTRODUÇÃO

Nesse documento vamos apresentar a arquitetura implementada no Trabalho Final de Arquitetura de Computadores, as otimizações que foram desenvolvidas e os programas montados com base nas requisições presentes na [Descrição do Trabalho](#).

1. Estrutura da Arquitetura Original e suas limitações

A arquitetura que nos foi apresentada no início do trabalho segue um padrão simples de 32 bits com a presença de 9 registradores. Sua configuração se assemelha figura 1.1 e seus microprogramas em firmware poderiam ser escritos seguindo a tabela na figura 1.2

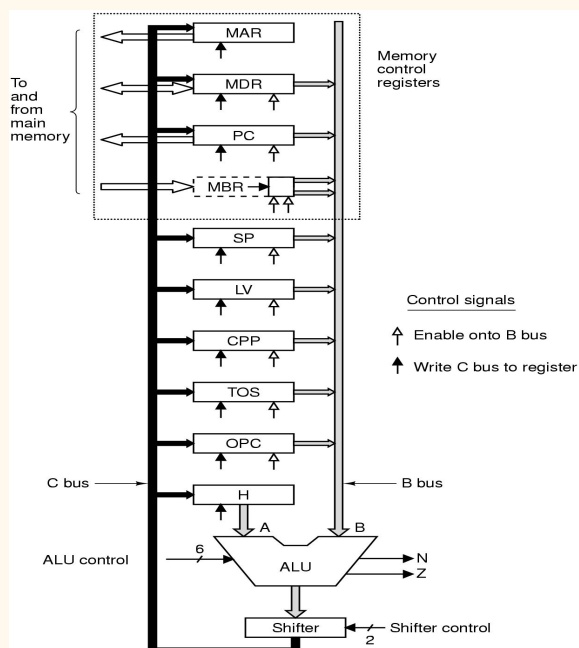


Figura 1.1 - Desenho da arquitetura.

NEXT ADDRESS										JUMP			ULA								BARRAMENTO C					MEMORY				BAR B																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
										JMPC	JAMN	JAMZ	SLL8	SLL9	F0	F1	ENA	ENB	INVA	INC	MAR	MDR	PC	X	Y	H	W	R	F																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													

JUMP		BARRAMENTO B		MEMORY	
001	JAMZ - SE O VALOR DO Z FOR 1, PULA PARA A PROXIMA INSTRUÇÃO + 256	000	MDR	001	FETCH
010	JAMN - SE O VALOR DO N FOR 1, PULA PARA A PROXIMA INSTRUÇÃO + 256	001	PC	010	READ
100	JAMPC - PULA PARA O VALOR DA PRÓXIMA INSTRUÇÃO APLICADA A UM "OU" COM O VALOR DO MBR	010	MBR	100	WRITE
		011	X		

Figura 1.1 - Tabela para escrita do firmware

```

1  from array import array
2
3  firmware = array('Q', [0]) * 512
4
5  # X = X + memory[address]
6
7  ## 2: PC <- PC + 1; fetch; goto 3
8  firmware[2] = 0b000000011_000_00110101_001000_001_001
9
10 ## 3: MAR <- MBR; read_word(MAR); goto 4
11 firmware[3] = 0b000000100_000_00010100_100000_010_010
12
13 ## 4: H <- MDR; goto 5
14 firmware[4] = 0b000000101_000_00010100_000001_000_000
15
16 ## 5: X <- H + X; goto 0
17 firmware[5] = 0b000000000_000_00111100_000100_000_011

```

Figura 1.3 - Exemplo de instrução de firmware na Arquitetura Original

O programa apresentado é capaz de realizar funções básicas de adição e subtração e já possui em seu firmware comandos pré-definidos dessas operações além de mov (memory[address] = x) e jz (if x == 0, goto address).

Contudo, por meio de uma breve análise, podemos verificar algumas limitações na arquitetura:

1. Não é possível realizar operações diretamente entre dois registradores a não ser o H (arquitetura possui apenas o barramento B). Isso faz com que operações simples como adição levem um passo a mais do que o necessário, visto que precisamos enviar os dados para o H antes de realizar qualquer operação. Comprova-se na função read_regs:

```

1  def read_regs(reg_num):
2      global MDR, PC, MBR, X, Y, H, BUS_A, BUS_B
3
4      BUS_A = H # BUS_A fica limitado ao H

```

Figura 1.4 - BUS_A é restrito ao registrador H

2. As instruções de firmware são limitadas ao X: não possuem instruções de firmware que modificam o registrador Y. Isso eleva o grau de complexidade da programação para a máquina.
3. Não possuímos circuitos de multiplicação e divisão na ULA. Logo, qualquer operação de multiplicação levaria a sucessivas adições e de divisão a sucessivas subtrações, resultando em um número excessivo de passos.

Portanto, considerando essas possíveis implementações, concluímos que as seguintes alterações seriam adequadas para otimização do computador apresentado:

1. Adição do barramento A: possibilitando uma fluidez na operação entre registradores e descartando a obrigatoriedade do movimento para o registrador H.
2. Adição de microinstruções em firmware para modificação do registrador Y: diminuindo o número de passos nas operações matemáticas e a complexidade na escrita de programas para a arquitetura.
3. Adição dos circuitos de multiplicação e divisão na ULA.

Explicamos cada implementação abaixo:

2. Alterações e otimizações

Adição do barramento A

Como vimos anteriormente nas figuras 1.1 e 1.4, não era possível fazer a seleção do registrador que iria para o BUS_A, sendo restrito ao H. Isso levava a um passo obrigatório de escrita no registrador H para cada operação. Logo, buscamos otimizar isso por meio da adição do barramento A à arquitetura, fazendo com que ela se assemelhe à figura 2.1.

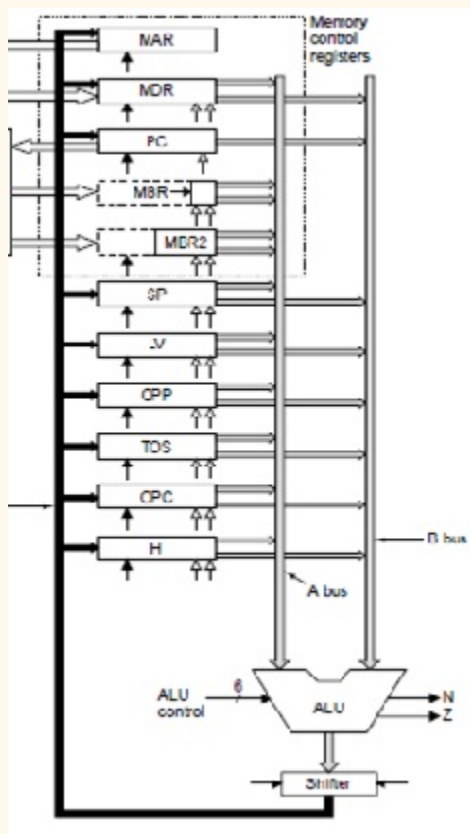


Figura 2.1 - Arquitetura que possibilita selecionar o barramento A e B

Para isso, seguimos os seguintes passos:

1. Adição da chave de seleção na função `read_regs` de leitura do registrador.

Para acrescentar o novo barramento é necessário tirar a restrição do BUS_A ao H (explicitado na figura 1.4) na função `read_regs` e possibilitar com que as nossas microinstruções em firmware selecionem quais registradores serão utilizados para as operações dentro da ULA (a e b). Para isso, adicionaremos 3 bits ao final de cada instrução do firmware (explicaremos no próximo passo). Depois disso, modificamos a função `read_regs` para a identificação de qual registrador irá para o BUS_A e qual irá para o BUS_B separando-os do comando por meio de uma simples operação. Segue demonstração:

```

1  def read_regs(reg_num):
2      global MDR, PC, MBR, X, Y, H, BUS_A, BUS_B
3
4      reg_num_a = reg_num & 0b111_000 # Pega apenas o barramento A - seleciona o BUS_A
5      reg_num_b = reg_num & 0b000_111 # Pega apenas o barramento B - seleciona o BUS_B
6
7      # Faz as verificações para pegar o valor correto de cada REG
8      if reg_num_a == 0:
9          BUS_A = MDR
10     elif reg_num_a == 1: #PC
11         BUS_A = 0 # Na micro arquitetura o BUS_A não recebe o PC
12     elif reg_num_a == 2:
13         BUS_A = MBR
14     elif reg_num_a == 3:
15         BUS_A = X
16     elif reg_num_a == 4:
17         BUS_A = Y
18     elif reg_num_a == 5:
19         BUS_A = H
20     else:
21         BUS_A = 0
22
23     if reg_num_b == 0:
24         BUS_B = MDR
25     elif reg_num_b == 1:
26         BUS_B = PC
27     elif reg_num_b == 2:
28         BUS_B = MBR
29     elif reg_num_b == 3:
30         BUS_B = X
31     elif reg_num_b == 4:
32         BUS_B = Y
33     else:
34         BUS_B = 0

```

Figura 2.2 - Nova função read_regs com seleção do BUS_A

2. Modificação das instruções de firmware

Para acrescentar o novo barramento, é necessário também modificar as instruções no firmware. As instruções originais possuíam apenas o endereço do barramento B (BUS_B) visto que o barramento A (BUS_A) era restrito ao H. Contudo, com a nova implementação se fez necessário adicionar 3 bits antes do endereço do BUS_B para endereçar o BUS_A. Segue figura 2.3 e compara-se com a instrução presente na figura 1.3.

Adição de microinstruções em firmware para modificação do registrador Y:

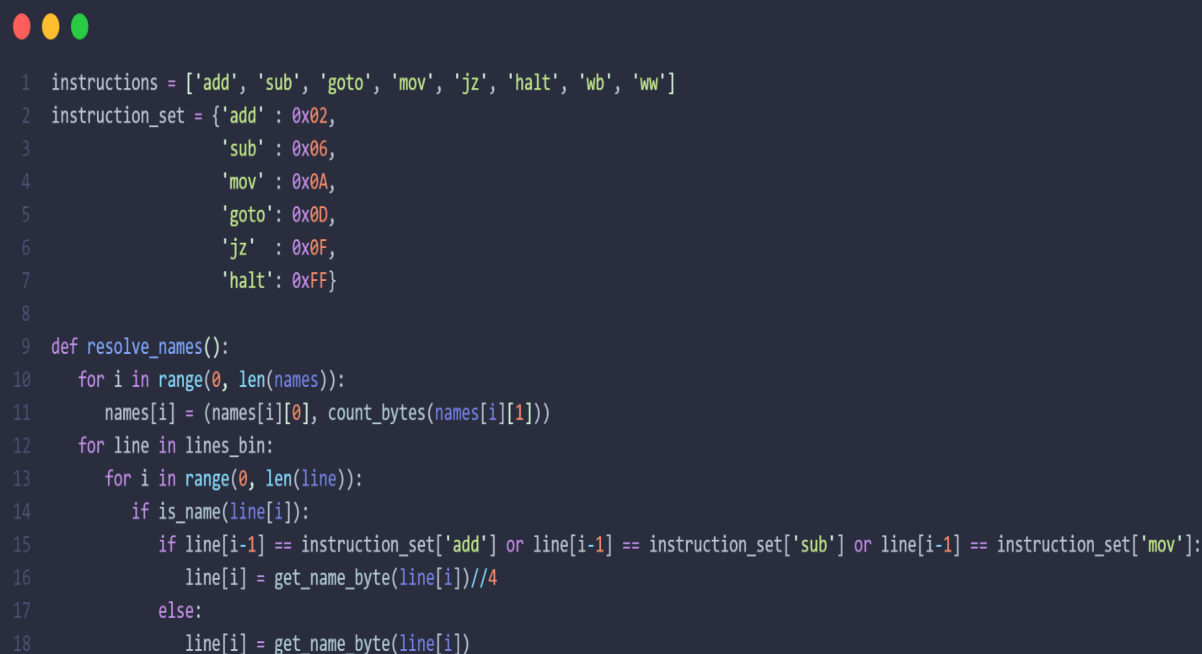
Como dito anteriormente, apenas operações com o registrador X eram implementadas no processador (arquivo ufc2x.py) e assembler (arquivo assembler.py). Seguem figuras 2.6 e 2.7.

```

1  # Operações com X
2
3  # X = X + memory[address] (add x, )
4
5  ## 2: PC <- PC + 1; MBR <- read_byte(PC) - fetch; goto 3
6  firmware[2] = 0b000000011_000_00110101_001000_001_000001
7
8  ## 3: MAR <- MBR; read_word(MAR); goto 4
9  firmware[3] = 0b000000100_000_00010100_100000_010_000010
10
11 ## 4: X <- MDR + X; goto 0
12 firmware[4] = 0b000000000_000_00111100_000100_000_000011
13
14
15 # X = X - memory[address] (sub x, )
16
17 ## 5: PC <- PC + 1; fetch; goto 6
18 firmware[5] = 0b000000110_000_00110101_001000_001_000001
19
20 ## 6: MAR <- MBR; MDR <- read_word(MAR); goto 7
21 firmware[6] = 0b000000111_000_00010100_100000_010_000010
22
23 ## 7: X <- X - MDR; goto 0
24 firmware[7] = 0b000000000_000_00111111_000100_000_000011
25
26 # memory[address] = X (mov x, )
27
28 ## 8: PC <- PC + 1; fetch; goto 9
29 firmware[8] = 0b00001001_000_00110101_001000_001_000001
30
31 ## 9: MAR <- MBR; goto 10
32 firmware[9] = 0b00001010_000_00010100_100000_000_000010
33
34 ## 10: MDR <- X; write; goto 0
35 firmware[10] = 0b000000000_000_00010100_010000_100_000011
36

```

Figura 2.6 - Firmware possuía apenas instruções com X



```

1 instructions = ['add', 'sub', 'goto', 'mov', 'jz', 'halt', 'wb', 'ww']
2 instruction_set = {'add' : 0x02,
3                   'sub' : 0x06,
4                   'mov' : 0x0A,
5                   'goto' : 0x0D,
6                   'jz' : 0x0F,
7                   'halt' : 0xFF}
8
9 def resolve_names():
10     for i in range(0, len(names)):
11         names[i] = (names[i][0], count_bytes(names[i][1]))
12     for line in lines_bin:
13         for i in range(0, len(line)):
14             if is_name(line[i]):
15                 if line[i-1] == instruction_set['add'] or line[i-1] == instruction_set['sub'] or line[i-1] == instruction_set['mov']:
16                     line[i] = get_name_byte(line[i])//4
17             else:
18                 line[i] = get_name_byte(line[i])

```

Figura 2.7 - No assembler também não havia a possibilidade de compilar operações com o Y. (Verifica-se no set de instruções e na função `resolve_names` que serão alterados)

Nesse contexto, para facilitar a implementação de alguns programas e tornar a programação na máquina mais otimizada e fluída, adicionamos os comandos de firmware para o registrador Y se inspirando nos que já estavam presentes do X e adicionamos ao assembler a compilação dessas instruções. Modificamos essencialmente os sets de instruções e a função `resolve_names`.

Nos mesmos programas foram adicionados instruções de multiplicação e divisão por conta do aprimoramento da ULA por meio de novos circuitos que será explicada posteriormente.

```

1  # Operações com Y
2
3  # Y <- Y + memory[address] (add y, )
4
5  ## 27: PC <- PC + 1; MBR <- read_word(PC) - fetch; goto 28
6  firmware[27] = 0b00011100_000_00110101_001000_001_000001
7
8  ## 28: MAR <- MBR; MDR <- read_word(MAR); goto 29
9  firmware[28] = 0b00011101_000_00010100_100000_010_000010
10
11 ## 29: Y <- Y + MBR; goto 0
12 firmware[29] = 0b00000000_000_00111100_000010_000_000100
13
14 # Y <- Y - memory[address] (sub y, )
15
16 ## 30: PC <- PC + 1; MBR <- read_word(PC) - fetch; goto 31
17 firmware[30] = 0b00011111_000_00110101_001000_001_000001
18
19 ## 31: MAR <- MBR; MDR <- read_word(MAR) - read; goto 32
20 firmware[31] = 0b00100000_000_00010100_100000_010_000010
21
22 ## 32: Y <- Y - MDR; goto 0
23 firmware[32] = 0b00000000_000_00111111_000010_000_000100
24
25 # memory[address] = Y (mov y, )
26
27 ## 33: PC <- PC + 1; MBR <- read_word(PC) - fetch; goto 34
28 firmware[33] = 0b00100010_000_00110101_001000_001_000001
29
30 ##34: MAR <- MBR; MDR <- read_word(MAR) - read; goto 35
31 firmware[34] = 0b00100011_000_00010100_100000_010_000010
32
33 ##35: memory[address] = Y; goto 0
34 firmware[35] = 0b00000000_000_00010100_010000_100_000

```

Figura 2.8 - Instruções no firmware para o registrador Y

```

instructions = ['add', 'sub', 'mov', 'goto', 'jz', 'wb', 'ww', 'mult', 'div', 'halt']
instruction_set = { # Endereços no firmware de cada instrução (do X e do Y)
    'add' : {'x': 0x02, 'y': 0x1B},
    'sub' : {'x': 0x05, 'y': 0x1E},
    'mov' : {'x': 0x08, 'y': 0x21},
    'goto': 0x0B,
    'jz'  : {'x': 0x0D, 'y': 0x24},
    'mult': {'x': 0x12, 'y': 0x26},
    'div' : {'x': 0x15, 'y': 0x29},
    'halt': 0xFF,
}

def resolve_names():
    for i in range(0, len(names)):
        names[i] = (names[i][0], count_bytes(names[i][1]))
    for line in lines_bin:
        for i in range(0, len(line)):
            if is_name(line[i]):
                if ( # Seleção da instrução para o x ou para o y
                    line[i-1] == instruction_set['add']['x'] or line[i-1] == instruction_set['sub']['x']
                    or line[i-1] == instruction_set['mov']['x'] or line[i-1] == instruction_set['mult']['x']
                    or line[i-1] == instruction_set['div']['x']
                    or
                    line[i-1] == instruction_set['add']['y'] or line[i-1] == instruction_set['sub']['y']
                    or line[i-1] == instruction_set['mov']['y'] or line[i-1] == instruction_set['mult']['y']
                    or line[i-1] == instruction_set['div']['y']
                ):
                    line[i] = get_name_byte(line[i])//4
            else:
                line[i] = get_name_byte(line[i])

```

Figura 2.9 - Instruções para o Y adicionadas ao Assembler

Dessa forma, conseguimos construir programas em assembly que utilizam dois registradores.

Adição dos circuitos de multiplicação e divisão na ULA

Com o intuito de melhorar o desempenho dos nossos programas em Assembly - retirando a função de multiplicação em loop-, adicionamos as operações de multiplicação e divisão da linguagem Python, respectivamente ‘*’ e ‘//’ na função da ULA e seus respectivos control_bits.

```

1 elif control_bits == 0b100000: #32
2     o = a * b # Circuito adicionado de multiplicação
3 elif control_bits == 0b100001: #33
4     o = a // b

```

Figura 2.10 - Adição das novas operações de multiplicação e divisão

```

1 instructions = ['add', 'sub', 'mov', 'goto', 'jz', 'wb', 'ww', 'mult', 'div', 'halt']
2 instruction_set = { # Endereços no firmware de cada instrução (do X e do Y)
3     'add' : {'x': 0x02, 'y': 0x1B},
4     'sub' : {'x': 0x05, 'y': 0x1E},
5     'mov' : {'x': 0x08, 'y': 0x21},
6     'goto': 0x0B,
7     'jz'  : {'x': 0x0D, 'y': 0x24},
8     'mult': {'x': 0x12, 'y': 0x26},
9     'div' : {'x': 0x15, 'y': 0x29},
10    'halt': 0xFF,
11 }

```

Figura 2.11 - Instruções para a multiplicação e divisão adicionados ao Assembler

```

1 def resolve_names():
2     for i in range(0, len(names)):
3         names[i] = (names[i][0], count_bytes(names[i][1]))
4     for line in lines_bin:
5         for i in range(0, len(line)):
6             if is_name(line[i]):
7                 if ( # Seleção da instrução para o x ou para o y
8                     line[i-1] == instruction_set['add']['x'] or line[i-1] == instruction_set['sub']['x']
9                     or line[i-1] == instruction_set['mov']['x'] or line[i-1] == instruction_set['mult']['x']
10                    or line[i-1] == instruction_set['div']['x']
11                    or
12                    line[i-1] == instruction_set['add']['y'] or line[i-1] == instruction_set['sub']['y']
13                    or line[i-1] == instruction_set['mov']['y'] or line[i-1] == instruction_set['mult']['y']
14                    or line[i-1] == instruction_set['div']['y']
15                ):

```

Figura 2.12 - Instruções para a multiplicação e divisão adicionados ao Assembler

Dessa forma, para adicionar essas operações à nossa arquitetura, tivemos que adaptar a nossa ULA (Unidade Lógica Aritmética). Adicionamos a ela o circuito de multiplicação, o qual pode ser visto abaixo, nas figuras 2.13 e 2.14.

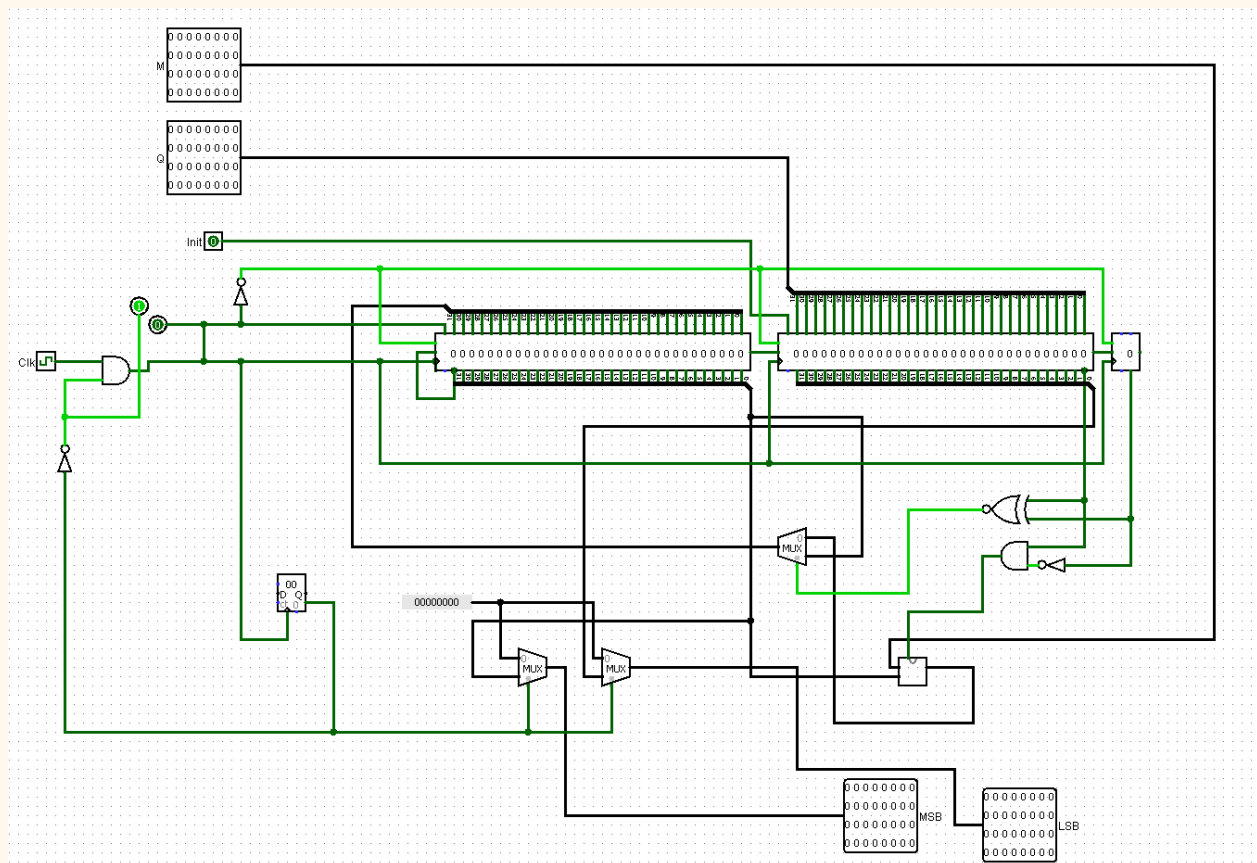


Figura 2.13 - Circuito de multiplicação no Logisim. Acrescentada nos arquivos do trabalho

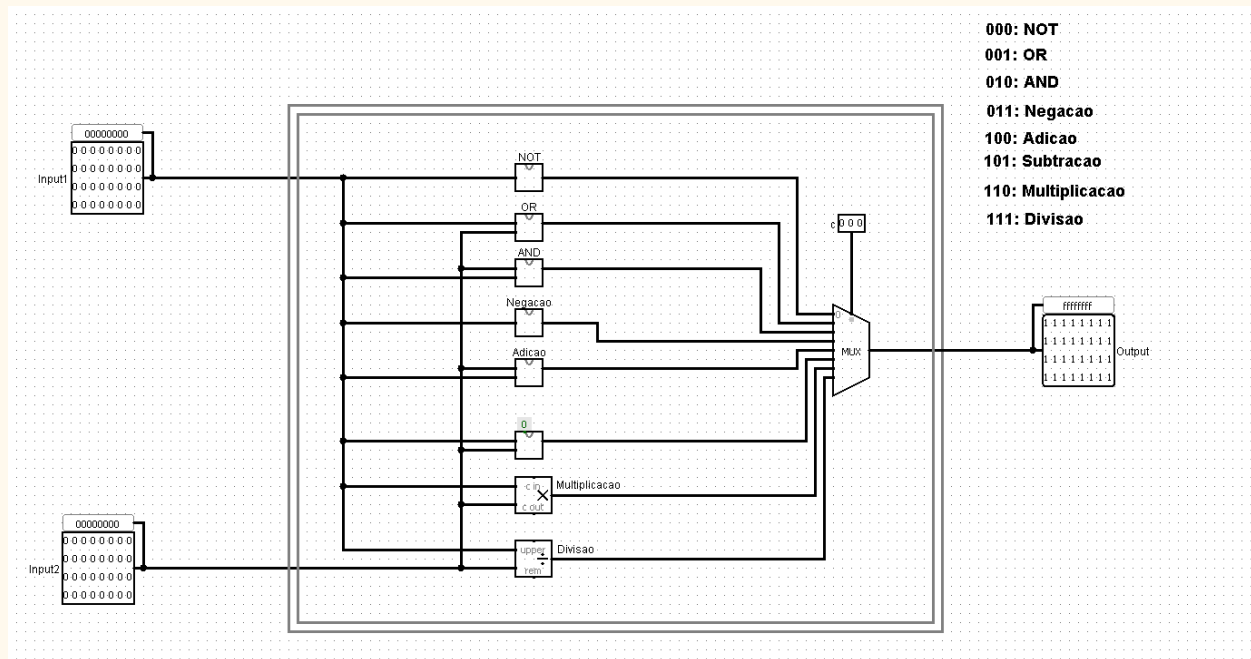


Figura 2.14 - ULA (Unidade Lógica Aritmética) no Logisim. - Acrescentada nos arquivos

Adicionar um circuito de multiplicação e divisão à Unidade Lógica Aritmética (ULA) de uma arquitetura de computador pode trazer benefícios significativos para a parte de software do sistema. Aqui estão alguns aspectos relevantes:

1- Com a inclusão de circuitos dedicados de multiplicação e divisão na ULA, as operações aritméticas podem ser executadas mais rapidamente, resultando em tempos de execução reduzidos.

2- Otimização de Cálculos Complexos: Muitos algoritmos e programas requerem operações de multiplicação e divisão para realizar cálculos complexos. Com a adição dos circuitos dedicados, esses cálculos serão executados de maneira mais eficiente e rápida.

Em suma, ao incorporar circuitos de multiplicação e divisão à ULA de uma arquitetura de computador, o software que faz uso dessas operações pode se beneficiar com um desempenho aprimorado, tempos de execução reduzidos, otimização de cálculos complexos, agilidade em aplicações específicas, redução de sobrecarga de software e aceleração de algoritmos dependentes dessas operações

3. Programas desenvolvidos:

CSW

```

1      goto main    # Questão 3 - CSW(a,b,c)
2      wb 0
3
4      r ww 1        # Resultado - Retorno da função
5      a ww 250      # Argumento da função CSW
6      b ww 250      # Operando da função CSW
7      c ww 249      # Argumento da função CSW
8
9
10     main add x, a   # Se a != c; a <- c; return 1 // x recebe o valor de a (x = x + a)
11         sub x, c    # x = x - c
12         jz x, else  # se x = 0 (a == c) vai para a linha else
13         add y, c    # y = y + c
14         mov y, a    # passa o y (que possui o valor de c) para a - memory[a] = y
15         halt       # finaliza
16
17
18     else mov x, r    # a == c; c <- b; return 0 // x = 0 -> passa o 0 para o r (retorna 0)
19         add x, b    # x = x + b
20         mov x, c    # passa o x para o c
21         halt       # finaliza

```

Comandos para compilar e rodar o CSW:

```
python3 .\assembler.py .\asm\questoes\csw.asm .\asm\bin\csw.bin
```

```
python3 .\run_program.py .\asm\bin\csw.bin
```


Fatorial

```

1      goto main # Questão 4 - Fatorial de um número
2      wb 0
3
4  r    ww 0      # Retorno da função - resultado do fatorial
5  c    ww 12     # Paramtro que terá seu fatorial calculado
6  u    ww 1      # Auxiliar que servirá para decrementar o número e multiplicar
7
8  main add x, c   # Bota o valor de c em x (x = x + c)
9      jz x, finalmain # Se N igual a 0, finaliza o programa e retorna r = 1
10     sub x, u
11     jz x, finalmain # Verificao se é 1
12     add x, u       # Volta o x no estado anterior
13     mov x, r       # Bota o valor de x em r (memory[r] = x)
14     add y, r       # Bota o valor de r em y (y = y + r)
15     goto loop      # Entra no loop de fatorial
16 loop sub x, u     # Subtrai 1 de x (x = x - u)
17     jz x, final    # Se x == 0 -> vai pro final e termina o programa
18     mov x, r       # Bota o valor de x (decrementado) em r (memory[r] = x)
19     mult y, r      # Multiplica y com r (y = y * r)
20     goto loop      # Volta para o loop
21 final mov y, r    # Ao final -> move o valor de y para o r (memory[r] = y)
22     halt          # Comando de parada
23
24 finalmain add y, r # Usa o Y para verificar se o valor inserido é 0
25         jz y, caso1
26         mov y, r
27         halt
28 caso1 add y, u
29         mov y, r
30         halt

```

Comandos para compilar e rodar o Factorial

```
python3 .\assembler.py .\asm\questoes\factorial.asm .\asm\bin\factorial.bin
```

```
python3 run_program.py .\asm\bin\factorial.bin
```

Potência

```

1  goto main          # Questao 1 - funcao potencia
2  wb 0
3
4  r ww 0             # Retorno da funcao potencia
5  m ww 5             # Argumento da potencia: base
6  n ww 3             # Argumento da potencia: expoente
7  a ww 0             # Argumento da funcao multiplicacao
8  b ww 0             # Argumento da funcao multiplicacao
9  z ww 0             # Retorno da funcao multiplicacao
10 u ww 1             # +1 or -1
11
12 main add x, m       # Bota x em m
13     jz x, caso1     # Se for 0, vai para caso 1
14     sub x, u        # Subtrai 1 de x
15     jz x, caso2     # Se for 0, vai para caso 2
16
17     add y, n        # Adiciona n em y: expoente para o Y
18     jz y, caso3     # Se y (n) for 0, vai para caso 3
19     sub y, u        # Subtrai 1 de Y
20     jz y, caso4     # Se for 0 depois (n era 1), vai para caso 4
21
22     mov y, n        # Bota n (expoente) para o Y novamente
23     sub y, n
24
25     add x, u
26     mov x, a
27     sub x, a
28
29     goto potencia
30
31 caso1 halt         # Base = 0, retorna 0
32 caso2 add x, m      # Base = 1, retorna 1
33     mov x, r
34     halt
35 caso3 add y, u      # Se expoente = 0, retorna 1
36     mov y, r
37     halt
38 caso4 add y, m      # Se expoente = 0, retorna a base
39     mov y, r
40     halt
41
42 potencia add x, m   # Faz a preparacao da potencia -> adiciona a base ao x
43     add y, n        # Adiciona o expoente ao y
44     mov x, r        # Move a base para o x
45     goto loop       # Manda para o Loop
46
47 loop jz y, final    # Se y (qtd de vezes que a potencia executou for = expoente no inicio) vai para o final
48     mult x, r        # Multiplica a base por r
49     sub y, u        # Subtrai 1 de y
50     goto loop
51
52 final mov x, r      # Move r para x
53     halt            # Finaliza
54

```

Comandos para compilar e rodar o Potencia:

```
python3 .\assembler.py .\asm\questoes\potencia.asm .\asm\bin\potencia.bin
```

```
python3 run_program.py .\asm\bin\potencia.bin
```

Conclusão e resultados:

Com as implementações e otimizações acrescentadas, conseguimos uma arquitetura mais rápida e eficaz no cumprimento de funções, obtendo os seguintes resultados nas questões desenvolvidas:

- Fatorial de 5: 111 passos
- CSW:
 - 22 passos se 1
 - 27 passos se 0
- Potência de 5^3 : 101 passos

Um desempenho significativamente melhor do que o que a arquitetura inicial apresentava.